DISSERTATION

# MULTILEVEL SECURE DATA STREAM MANAGEMENT SYSTEM

Submitted by

Xing Xie

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2013

Doctoral Committee:

Advisor: Indrakshi Ray

Indrajit Ray
Robert France
Daniel Turk

ABSTRACT

MULTILEVEL SECURE DATA STREAM MANAGEMENT SYSTEM

With the advent of mobile and sensor devices, situation monitoring applications are now feasible. The data processing system should be able to collect large amount data with high input rate, compute results on-the-fly and take actions in real-time. Data Stream Management Systems (DSMSs) have been proposed to address those needs. In DSMS the infinite input data is divided by arriving timestamps and buffered in input windows; and queries are processed against the finite data in a fixed size window. The output results are updated by timestamps continuously. However, data streams at various sensitivity levels are often generated in monitoring applications which should be processed without security breaches. Therefore current DSMSs cannot prevent illegal information flow when processing inputs and queries from different levels.

We have developed multilevel secure (MLS) stream processing systems that operate input data with security levels. We've accomplished four tasks include: (1) providing formalization of a model and language for representing secure queries, (2) investigating centralized and distributed architectures able to handle MLS continuous queries, and designing authentication models, query rewriting, optimization mechanisms, and scheduling strategies to ensure that queries are processed in a secure and timely manner, (3) developing query sharing approaches to improve quality of service. Besides we've implemented extensible prototypes with experiments to compare performance between different process strategies and architectures, (4) and proposing an information flow control model adapted from the Chinese Wall policy that can be used to protect against sensitive data disclosure, as an extension of multilevel secure DSMS for stream audit applications.

# ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my advisor, Dr. Indrakshi Ray for her patiently support and continually encouragement during my PhD study at Colorado State University. With her insightful advice, my skills on writing and communications, as well as publication records are improved day by day. Besides the academic guidance, Dr. Ray has shared her study and life experience that give me confidence and ideas to cope with the challenges in life.

I would like to thank Dr. Indrajit Ray for kindness help in these years and teaching me computer security theories which is critical aspect of this dissertation. I would like to thank Dr. Daniel E. Turk for contributing insightful comments in my proposal and preliminary presentations. I would like to thank Dr. Robert France for teaching the knowledge of software models and engineering, which helps a lot during prototype development in my dissertation. I also want to thank Russell Wakefield, I learn from him for the earnest and responsible attitude to class and students. My thanks also to Dr. Ping Yang from Stony Brook University and Dr. Raman Adaikkalavan from Indiana University South Bend, I've improved programming and writing skills from you.

I would like to thank all faculty members and colleagues in Computer Science department, for giving me the advices and conveniences in my daily life. Thanks to all my friends and colleagues here, Yun Zou, Aritra Bandyopadhyay, Tarik MoatazRamadan Abdunabi, Malgorzata Urbanska, Dieudo Mulamba, and Zhiquan Sui, for their constructive criticism and support.

To my mother Fei, thank you for the determined mind supporting my PhD dream in U.S. even through you are suffering the disease. You are always in my heart and sharing with my pains and gains. To my father Xingyu, thank you for showing me how to be positive and tough in life under challenges. My special thanks to my lifelong friends Gail Woods and Porter Woods, you welcome

and help me enjoy life here.

Last but not least, I would like to express my gratitude to my wife Zhaohua for her deepest love and wholeheartedly support. Thank you for believing in me right from the beginning of this journey. You are truly wonderful.

# DEDICATION

*This thesis is dedicated to my mother Fei,*

*to my father Xingyu,*

*and to my love Zhaohua.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# Introduction

## 1.1 Introduction to Data Stream Management System

Over 40 years development, relational DataBase Management System (DBMS) is sufficient to process one-time queries against finite pre-stored relations with sound mechanisms like query optimization, crash recovery, security enforcement. On the other hand, with the advancements of mobile devices and data transmission speed, situation monitoring applications such as border security monitoring, battlefield monitoring, stock marketing analysis, emergency control and threat monitoring, are becoming a reality. The data processing system should be able to collect large amount data with high input rate, compute results on-the-fly and take actions in real-time. To enable real-time stream processing, there are eight requirements should be satisfied which are demonstrated by Stonebraker and his fellows in [63]. The traditional DBMS cannot be used directly for such applications because of dissatisfaction at five critical requirements:

1. The system should be active to process messages “in stream” without any requirements to store them. Traditional DBMS is passive system which stores the input first then process, which causes high latency.

2. Queries should use SQL on streams with built-in extensible stream-oriented primitives and operators. Traditional SQL does not support stream-specific queries.

3. The system should be able to handle unexpected streaming input conditions such like de-

layed, out-of-order, and missing. In DBMS extra mechanism needs to be developed for those imperfect date and unexpected input rate.

4. The output of stream processing must be predictable and repeatable. Since SQL queries are one-time query, they are only required to be repeatable after execution. Besides DBMS is insufficient to ensure predictable and deterministic execution semantics for stream-specific queries.

5. The system should be able to handle queries on combinations of live streaming data and store tables. Some business applications perform seamlessly data analysis starting at some point from the past data, then catch up to the real time. Such functions are not supported in DBMS.

6. The system should be scalable and available at all time, and the integrity of data should be guaranteed despite failures. Distributed DBMS successfully satisfies such requirement.

7. Distributed systems automatic and transparent to users. Similarly DBMS with the distributing extension meets this requirement.

8. Stream processing system is highly QoS-oriented. Commercial DBMSs in these days are equipped with optimized and minimal-overhead engines to handle those real-time computations on large amount of inputs.

To fulfil those missing stream processing requirements as well as address the stream processing applications, Data Stream Management Systems (DSMSs) [8, 12, 15, 30, 46, 64, 66, 41] have been proposed. The infinite input data is divided by arriving timestamps and buffered in input windows;

and queries are processed against the finite data in a fixed size window. The output results are updated by timestamps continuously.

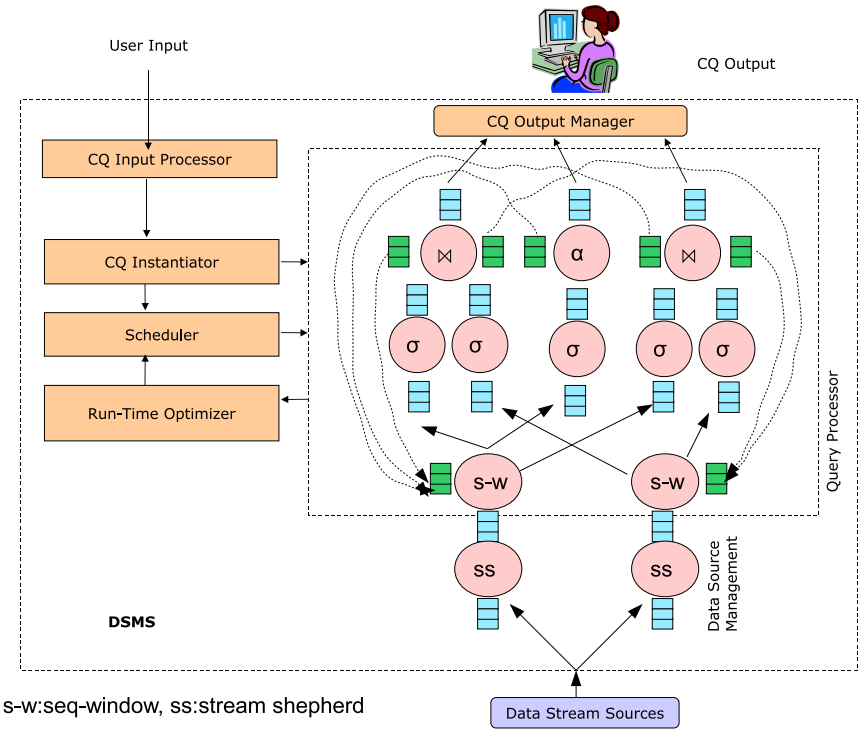


Figure 1.1: Data Stream Management System (DSMS)

A DSMS [12, 23, 26] architecture (based on the STREAM system [8]) is shown in Figure 1.1. The Continuous Query (CQ) can be defined by specification languages, then processed by the input processor to generate a *query plan*. Each query plan is a directed graph of operators like select, join, aggregate, etc. Each operator is associated with one or more input *queues* and an output queue. Those queues are used by the operators to propagate tuples. *Synopses* are temporary storage structures used by the operators (e.g., join) that need to maintain a state. One or more synopses are associated with each operator that needs to maintain the current state of the tuples for future evaluation of the operator. The generated query plans are then instantiated, and query operators are put in the ready state so that they can be executed. Based on stream scheduling strategies,

the scheduler can pick a query, an operator, or a path as the scheduling unit for execution. The run-time optimizer monitors the system, and initiates load shedding mechanisms as and when required. Both these QoS delivery mechanisms minimize resource usage (e.g., queue size) and maximize performance and throughput. In addition, other QoS improvement mechanisms such as static and dynamic approximation techniques are used to control the size of synopses. All the input tuples are first processed by the Data Source Manager, which enqueues the tuples to input queues of *all* the leaf operators associated with the stream. In the directed graph of operators, which is named *operator tree*, the data tuples are propagated from the bottom most leaf operator to the root operator. Each operator produces a stream of tuples. After a processed tuple exits the query plan, the output manager sends it to the query issuer.

## 1.2 Problem Description and Motivation

Often times, the input data in real-time monitoring applications involve data streams belonging to different security levels. Since database system processes personal and confidential data, privacy preservation and security control are necessary. For example, a soldier equipped with sensors sending out health and position data periodically can be accessed by the commander while the medic is only allowed to access the health info. In DSMS, users in different security classifications access and share a database consisting of a variety of sensitive data.

There are three major requirements for database security [17]: confidentiality, integrity, and availability. Under the context of stream processing applications, arrival data is used mainly for continuously observation, analysis and quick response, rather than long-term storage. So the confidentiality and availability are the two issues in secure DSMS development. Researchers have worked on secure query processing on DSMSs with access control. Specifically, these secure

DSMS works [4, 21, 20, 50, 54, 55] focus on providing forms of role-based access control where users are assigned to roles, roles are assigned to permissions, and users acquire permissions by activating the subset of roles assigned to them. However, current RBAC DSMSs are not perfect solutions for continuous query applications by our observations as following. First, the covert channel problem exists in RBAC DSMS architecture. A covert channel is a transfer of sensitive information from one process violates security policies, by the manipulation of a system resource in such a way that it can be detected by another process. Second, in order to prevent security breach, queries issued from users at different security levels should not communicate between each other by current DSMS access control policies. Such rigid isolation of query process eliminates the possibilities of sharing computations and storage resources across levels. Such barrier can be broken down if queries can be shared in a safe and effective manner. Besides, erroneous omission of an access control check may reveal confidential data. Integration of third party off-the-shelf software may cause policy checks to be bypassed altogether.

Besides access control model, security issues also exist in scheduling and load shredding methods used in DSMS. Current studies of DSMS performance concentrate on better scheduling [13, 45] and load shedding strategies [23, 60, 29] trying to optimize the memory and CPU usages. However, all those methods does not take security protection into account and cannot be directly applied to DSMS applications with sensitive data.

On the other hand, DSMS is performance-oriented. Even though many approaches have targeted on QoS with better scheduling and revising ideas, sharing execution and computation among queries are seldom explored. For example, queries submitted at different times by the same user or at the same time between different users are not supported in general DSMSs. Besides, earlier researches [37, 59, 28, 39] on sharing computation costs in DBMS cannot be directly used

in our research. Most approaches focus on optimizing join queries; but the join operations are implemented differently in data streams and database systems, so we cannot use many of these optimization techniques. Moreover, the queries in DBMS are not continuous and some of the proposed approaches apply to one-time queries only. Also, strategies that optimize multiple queries at any given point of time to find the best possible plan may not work in data stream systems as the queries arrive asynchronously. As a result, new sharing approaches should be developed to handle the special conditions in continuous queries. In addition, we need to prevent security violation since while sharing queries across different security levels.

To have a deep investigation into those issues, we are developing DSMSs with multilevel security (MLS) control. The motivation for this is that MLS systems with its centrally-defined labels have very simple and well-understood information flow policies. Compared with RBAC systems, ours is a simplified and complete system which bears all need-to-solve security and performance issues described above. Our two main goals are to find solutions to prevent illegal information flows in MLS-DSMS applications, and explore the possible sharing mechanisms between queries across different levels without security breaches. Experiments on centralized and distributed prototypes are conducted to find the trade-offs of MLS security enforcement and process sharing. We also explore the feasibility of applying MLS to distributed environment, as well as integrating new access control mechanisms such as Chinese Wall policies in DSMS.

This research work is significant. To our best knowledge, this research is the first work applying multilevel security control to DSMS. The MLS-DSMS formalization model can express the security level during query specification and query processing. The new scheduling strategies, which prevent overt and covert channels during multilevel queries processing, can also be used for network security research. The approaches of query sharing can be applied to not only streaming

applications but also traditional DBMS. The experiments of prototype implementation can provide statistical results of the overheads by introducing the MLS mechanism, and the benefits using sharing and MLS-specific scheduling strategies. The investigations of distributed network and adapting new access control policies such as Chinese Wall [58] bring forward ideas of applying the secure DSMS to cloud applications.

The challenges of MLS-DSMS development include two main aspects from stream management system and MLS control respectively.

- Some of unique characteristics of data stream processing systems are: (1) the input characteristics of data streams are usually not controllable, highly bursty, continuous, and are typically unpredictable, (2) data streams are read-only, (3) raw data streams are generated by stream sources and derived data streams are generated by query operators, (4) data streams are shared between operators to minimize resource usage, (5) queries are long running and are not snapshot queries, (6) queries can involve data streams and relational tables, and (7) applications have quality of service and accuracy requirements.

- Some of the unique requirements of multilevel security as well as other access controls are: (1) system elements are classified via security levels, (2) prevention of covert storage and timing channels, (3) trusted components vs. untrusted components, (4) overhead at each component of the underlying system, (5) and under distributed network, how to preserve security control and effective scheduling.

## 1.3 Research Tasks

To address the above challenges, we summarize the research goals as four tasks. These tasks are cohesive and related to each other, for the major goal of developing multilevel secure DSMS streams with illegal information prevention and better performance in terms of faster execution time. In general the following four tasks will be performed:

**Task 1: Formalizing a Model and Language for Processing MLS Continuous Queries:** We will develop a formal model for processing multilevel secure continuous queries and propose a language for expressing such queries. We plan to extend the Coninuous Query Language (CQL) [9] and propose a new semantics that is needed to process MLS continuous queries. This will help define the notion of equivalence between queries needed for query plan optimization and sharing.

**Task 2: Investigating Centralized and Distributed MLS-DSMS Architectures:** We will explore the possible DSMS architecture designs are able to address MLS continuous queries: Centralized system such as replicated and trusted MLS-DSMSs, and a simple distributed system with load balancing algorithm. To ensure secure execution for each architecture, we plan to (1) identify the trusted components including input stream shepherd operator, query plan generator, query processor and so on, (2) introduce authentication modules with authorization check to prevent illegal information flow, (3) design secure scheduling mechanisms, (4) and develop safe load distribution algorithm in distributed system.

**Task 3: Designing Sharing Approaches between Queries in the Same or across Different Levels:** DSMS expects heavy load of multiple queries and bursty inputs during execu-

tion. Without security violations, sharing queries as many as possible is the straightforward method to improve QoS by reducing execution time. Sharing queries in same level has been published in our work [5, 6]. Besides that we are presenting the sharing possibilities across different levels in more complex cases in this dissertation.

In addition to the three tasks, prototypes of replicated, trusted and a simple distributed architectures will be implemented. The three prototypes allow us to study the effects of the different architectures and process strategies on the performance for processing typical MLS continuous queries. There are two main factors we would like to investigate: (1) The secure enforcement overheads by the new trusted processors, which include the running time of user authentication as well as the extra scheduling efforts for secure executions. (2) The performance gain and differences from the sharing query processing results, which means how much response time we can reduce via reusing existing query processing results. We will use different kinds of MLS queries like select, aggregation, and join for a complete overview on the overhead and performance gains via experiments.

**Task 4: Proposing CW-DSMS an Information Flow Control Model Adapted from the Chinese Wall Policy:** In the near future, clouds will provide situational monitoring services using streaming data. Offering such services require securely processing data streams generated by multiple, possibly competing and/or complementing, organizations. Processing of data streams also should not cause any overt or covert leakage of information across organizations. Reusing the architecture design and query processing mechanisms, in this dissertation we also propose an information flow control model adapted from the Chinese Wall policy that can be used to protect against sensitive data disclosure. This secure DSMS

extension is designed for stream data auditing applications.

## 1.4 Dissertation Structure

The dissertation is organized as follows. Chapter 2 presents related work. In Chapter 3, we give a background introduction on the design, architecture, and process mechanism of Vanilla Stanford STREAM, as well as discussions on the limitations of security preservation and performance issues under applications with sensitive information. Chapter 4 discusses the multilevel formalization model and continuous query language, and the extension to support level-specific queries. In Chapter 5 we first present the considerations of possible MLS-DSMS, then give details of replicated architecture which provides better performance using secure sharing approaches and secure execution via revised scheduling method. Chapter 6 discusses trusted architecture which provides more flexibility on sharing across different levels without security violation. In Chapter 7 we propose the ideas of applying DSMS to distributed environments, with exploration on topics of group construction, secure execution and load distribution. In Chapter 8 we present a stream audit DSMS using Chinese Wall policy access control. In Chapter 9, we first provide the MLS-DSMS implementation details on critical components, then present the experiment evaluations on the overhead and performance gains in MLS-DSMS. In Chapter 10, the conclusion and future work are discussed.

# Chapter 2

# Related Work

## 2.1 Real-time and Stream Processing Systems

**Temporal and Real-Time System Developments:** Applications involving time-related input data appear in works related to temporal and real-time databases [56]. In temporal DBMS, input data come with arriving timestamps, so queries can be issued on data in certain time intervals. The idea of valid time computation inspires the window buffer processing in DSMS. However, temporal architecture cannot be adapted to DSMS applications because an extra temporal database is built for queries; and all queries and input data are predictable. For real-time systems, their theories cannot be used directly because of the differences on query duration, scheduling objects, and security threats between real-time and data stream systems. First, real-time DBMS deals with transient transactions while DSMS handles continuous queries. Second, real-time DBMS try to schedule isolated transactions while DSMS uses operators as the execution unit. The last, in order to cause a security breach, transactions might set up inference or covert channel via accessing the same data item while continuous queries try to manipulate the sharing response time.

On the other hand, there are researches focus on designing a real-time MLS DBMS where transactions having timing constraint deadlines executes in serialization order without security violations. Issues like security breach and task scheduling are similar to our MLS-DSMS development. Many concurrent control protocols, like 2PL high priority, OPT-Sacrifice, and OPT-WAIT [38], deal with the high level transactions by suspending or restarting them if they conflict with

low level transactions. However, the starvation on high level transactions becomes serious if there are too many conflicts in the system. S2PL [61] provides a better way on balancing the security and performance among conflicting transactions: high level transactions should wait for the commission of conflicting low level transactions only once then executed. Scheduling strategy in MLS real-time transaction processing must address security, serialization and transaction deadlines, whereas the MLS-DSMS must address security, query response time and throughput.

**Data Stream Management Systems:** Most of the work carried out in DSMSs addresses various problems ranging from theoretical results to implementing comprehensive prototypes on how to handle data streams and produce near real-time response without affecting the quality of service. There have been lot of works on developing QoS delivery mechanisms such as scheduling strategies [26, 11, 13, 45, 10, 22, 72, 27] and load shedding techniques [26, 67, 68, 33, 14, 48]. Some of the research prototypes include: Stanford Stream Data Manager [12], Aurora [15], Borealis [30], and MavStream [46]. MaxStream Project developed by ETH Zurich [18] redesigns DSMS as middle layer reusing popular existing stream processing systems.

Commercial DSMS products have been developed in these years such as IBM InfoSphere Systems ver.3.0 [40], StreamBase CEP [65] and webMethods Business Event [7]. The goal is to deploy for applications including algorithmic trading, market data management, intelligence and surveillance, risk (pre and post-trade) evaluation, smart order routing, transaction cost analysis, pricing and analytics, multi-asset trading, fraud detection, network monitoring, signal generation, statistic assistant, etc. Clients and partners include buy and sell side firms, global exchanges, intelligence and security organizations, eCommerce and online gaming firms, technology providers, and more. Those products are driven by complex event processing which aims to achieve better QoS stream applications.

## 2.2 DSMS Security

**Security Models**: There has been several recent works on RBAC secure DSMSs [4, 21, 20, 50, 54, 55]. The authors in [4] present a three-stage framework to enforce access control without introducing special operators, rewriting query plans, or affecting QoS delivery mechanisms. The framework moved access control enforcement outside the query processing, and allows user-level and role-level sharing of CQs and prevents underprivileged CQs from processing all tuples. We adapted the ideas of sharing between queries issued by users logged in same roles.

In punctuation-based enforcement of RBAC over data streams [54, 55], access control policies are transmitted every time using one or more security punctuations before the actual data tuple is transmitted. Query punctuations define the privileges for a CQ. Both punctuations are processed by a special stream shield operator that is part of the query plan. If the access check is successful, the data tuples that follow the punctuations are allowed to pass. However, this method expects input data within the same policies or in the same security level come in a consecutive way. The policy switching cost will be extremely high if input data in different levels come with random order. So punctuation approach is restricted to applications where input data in same policies are clustered and will be handled sequentially.

Borealis DSMS project in [50] uses a post-query filter to enforce access control policies. The filter applies security policies after query processing but before a user receives the results from the DSMS. The main drawback is keeping users connected to the system even though there is no output after post-filtering. Moreover, the access control filtering is done after query specification which introduces wasted computations. To reduce the cost, supporting RBAC via query rewriting techniques are proposed in [20, 21]. According to the privileges of the query submitter, queries are

checked against a policy map for authorization before execution. Our system development adapts their rewriting ideas by revising raw queries with level information for sharing analysis.

**Information Flow control:** MLS systems were the first to formalize the idea of information flow control across centrally-defined security classifications. Most of the work in this area assumed that the security labels cannot be changed inside the application. Myers and Liskov [53] proposed a decentralized information flow control model which allows the users to control the flow of their information and also allows for explicit declassification of information. Decentralized Information Flow Control (DIFC) gives users the ability to create new policies while remaining constrained by the information flow policies of others. Several researchers have worked on DIFC OS-level policies [36, 77, 49]. Creating a language to express DIFC policies have also been explored by researchers [35]. DIFC model provides threat detections on malicious data modification to prevent illegal information flow. The DIFC model can be the future work of MLS system development integrated with user-specified constraints.

Decentralized Event Flow Control (DEFC) [51] have been proposed an architecture for expressing event-flow security policy in distributed multi-domain applications. In DEFC model, events are classified with confidentiality and integrity labels which are processed by different processing units. A unified event dispatcher is responsible to distribute each event to isolated security-compatible unit. The unit finishes the computation then updates the event's security labels if applicable. Our research adapts two ideas from them: isolating processing units in the system to shut down potential unsafe communications, and providing output results with security level upgrading using least upper bound of all computation involving data levels. For example, level of aggregation output result should be the highest level of all input computing data.

On the other hand, the DEFC model cannot be applied to MLS -SMS because (1) The schedul-

ing unit is operator/plan in MLS systems rather than event. (2) Each event is isolated processing, while MLS-DSMS is able to share processing plans based on security labels and similar query context. (3) The units in event processing system are built with specific processing functions, while in MLS system the processing unit can be completely replicated except assigned with different security labels.

**Chinese Wall Policy:** Brewer and Nash [19] first demonstrated how the Chinese Wall policy can be used to prevent consultants from accessing information belonging to multiple companies in the same conflict of interest class. However, the authors did not distinguish between human users and subjects that are processes running on behalf of users. Consequently, the model proposed is very restrictive as it allows a consultant to work for one company only. Sandhu [58] improves upon this model by making a clear distinction between users, principals, and subjects, defines a lattice-based security structure, and shows how the Chinese Wall policy complies with the Bell-Lapadula model [16]. In this work we've implemented a DSMS prototype with Chinese Wall policy control and reusing design architecture and query processing mechanisms.

**Security Threats:** Imperva [42] a business security solution company proposed ten top security threats on commercial DBMS. There are five from ten related to database design fit to our secure DSMS system research:

1. Excessive privilege abuse. Users or applications are granted database access privileges in excess of "business need-to-know" privileges in excess of business need-to-know. For example, a teacher assistant can update the student scores but not their personal information.

2. Legitimate privilege Abuse. Users might abuse legitimate access privileges for unauthorized purposes, e.g., combines two authorized tables (e.g., Health record, Resident info) to con-

struct a big table can tell sensitive information (e.g., John lived in Denver had a heart attack two months ago).

3. Privilege Elevation. With database platform software vulnerabilities, hackers might be able to get access privileges as an administrator.

4. Denial of Service (DOS). Common DOS techniques include data corruption, network flooding, and server resource overload. Resource overload is particularly common in stream processing environments.

5. Weak Authentication. Weak authentication schemes might cause identity and login credential threats. If happens the hacker can deploy his own strategies to obtain sensitive information.

Our MLS-DSMS prototype implementations have addressed 2nd, 3rd and 4th threats via the following ways: (1) using security level as an attribute to grant query access control, (2) enforcing simple security property and the restricted $\star$-property of the Bell-Lapadula model in our multilevel security system for all users, (3) and propose distributed MLS-DSMS framework to make each node running as a server. Mitigation developments for 1st and 5th threats remain in our future work.

## 2.3 DSMS Performances

**Scheduling methods:** With high volume of unexpected inputs, the DSMS needs a effective scheduling method to run stream-specific queries in a long run. simplified and complex methods have been developed and described as following:

1. Round-Robin. This is the scheduler method run in Stanford STREAM system [8]. Each operator in plan will be scheduled in a linked list and will be run for a fixed time unit or input queue becomes empty. Round-Robin avoids starvation because in each round each operator will be executed for some time. There is no priority among tuples or operators.

2. FIFO First In First Out. DSMS executes input tuples through the plans based on the arrival timestamp. Operators cannot access next tuple until the current tuple is completely handled. As a result, some queries might suffer starvation if their input tuples in a bigger timestamp are buffered in the queue waiting for others to complete. This scheduling does not support query priority in MLS under bursty input situation. Since the timestamp used for ordering cannot be changed, FIFO cannot put execution priority to specific queries with better resource release but later input arrival. On the other hand, FIFO is free from security violation because execution order is based on tuple arrival timestamp which cannot be manipulated by users.

3. Greedy strategy. At any time instant, the operator has biggest operator memory release capacity *CO* will be selected to execute. *CO* means the maximum number of tuples can be consumed within this time unit by the tuple handling operator. However, the throughput of queries are low under large numbers of input and one operator with best *CO* will always be scheduled for execution while others are blocked.

4. Chain strategy [13]. At any time, DSMS considers all tuples that are currently arrived in the system. DSMS schedules a single time unit for the tuple that lies on the segment of consecutive operators with biggest segment memory release capacity. If there are multiple such tuples, system will pick the tuple which has the earliest arrival time. The scheduling

priority is determined by segment memory release capacity. However, covert channel can be established since high level plans always release more memory because of consuming more qualified input data that are not accessible to low-level queries.

5. Operator Path Capacity strategy [45]. It considers the processing rate of an operator path $P_i$ (processing capacity) $C^P_{P_i}$ as the priority. This method is an optimal one in terms of total tuple latency among all scheduling strategies. However it suffers the same problem as Chain strategy because in most cases the throughput of high level queries is much higher.

6. Segment strategy [44]. Instead of using operator path as priority unit, it first divide paths into segmentation then set up execution order among those pieces according to their segment processing capacity. It improves the memory requirements on path capacity strategy by sacrificing some response time for specific plan. Different from Chain, the execution priority is assigned to the operators in segment rather than the tuples. There are many segmentation methods on operator path. A simplified segment strategy can be applied where only two segments is used in each operation path. Since the leaf nodes (normally the selection and projection nodes) in a path have faster processing capacities and lower selectivity while others have much slower processing rate, the first segment includes leaf nodes and consecutive operators if their capacity reaches a fixed ratio (like 80%) of previous consecutive nodes. The second segment contains other operators in the path. The simplified segment strategy is one of Memory Optimal Segment (MOS) strategies which aims to minimize the total memory requirement as well as decrease the tuple latency.

None of these strategies can be directly applied to our system as they cause illegal information flow. So we are creating new MLS scheduling strategies can solve covert channel problem as well

as provide acceptable performance.

**Load Shedding:** Load shedding is another way to ensure QoS requirements of executing queries during bursty input by discarding some input tuples. Aurora project developers proposed that a proper load shedding mechanism should apply reasonable tuple-dropping algorithms on general queries [67] and aggregation operators [68] to reduce the relative error as well as satisfy the QoS requirements. The load shedding algorithms can be a random partial selection from all the input data, or integrate a semantic drop operator to some query plan according to fixed query coefficients like selectivity and Loss/Gain ratio (low-data-utility/CPU-cycle-saving). Besides, their conclusion forms the foundations of DSMS load shedding developments.

Based on researches [23, 23, 60, 29] load shedding mechanism should cover the following fundamental issues. First, what are the timings and conditions to activate load shedding by system semantics. For example, parameters like current memory usage, CPU-cycle rate and bandwidth are taken into account for load shedding from Aurora project [23]. Second, where is the perfect execution location in query plans for shredder operators. Authors in [60] claimed that function calls in input source operator is a best place for load shedding which saves buffering memory and processing time. On the other hand, if shedding happens on some input stream shared by multiple queries, the system must consider effects to those queries without shedding needs. Third, what is the proper quantity of shedding load such that the quality of query outputs is not compromised? The unpredictable input in DSMS reduces the efficiencies of pre-fixed shedding parameters, such as selectivity, potential data utility and CPU cycle gain. *Feedback control-based framework* implemented in Borealis project [29] is proposed to review output results periodically and then make shedding adjustments to fit the QoS better. However, Borealis FIFO scheduling method will cost huge buffer during brusty input data. Our future work is going to adapt the feedback framework

with a sophisticated memory-optimized scheduler.

**Query Sharing:** In the context of DBMS, researchers have investigated how queries can benefit by sharing their computation costs. Finkelstein [37] demonstrated how query graphs can be used for detecting common sub-expressions across multiple queries. Sellis [59] investigates the problem of multi-query optimization where the goal is to obtain a good plan for multiple queries. Chen and Dunham [28] have also looked into the problem of efficiently identifying common sub-expressions for processing multiple queries. Goldstein and Larson [39] focus on how queries can be optimized by using results from materialized views.

In general DSMSs like STREAM, Aurora, and Borealis, queries issued by the same user at the same time can share the Seq-window operators and synopses between each other. Besides common input source operators, sharing intermediate computation results is a better way to make big performance achievement. Jin and Carbonell [47] look into the problem of using predicate indexing for query optimizing in streams where not all the continuous queries are submitted at the same time. In this approach, a relation schema stores existing query plan information which will be compared and updated when a new query arrives. Our DSMS development adapts their idea of buffering query plans for comparison between existed and new queries.

## 2.4 Distributed DSMS

Cherniack et.al. [31] proposed the ideas of extending Aurora system to distributed environments. They presents critical issues and solutions on development of distributed stream processing system in their work.

Their proposed system $Aurora^*$ consists of multiple single-node Aurora servers that belong to the same administrative domain and cooperate to run the Aurora query network on the input

streams. The system is able to dynamically distribute the query load in terms of boxes. Specifically, the operators/boxes of query plan are dynamically distributed to different machines. $Aurora^*$ development raises several critical issues in our distributed DSMS development: (1) How to set up the distributed network and how to reduce the bandwidth of communications? (2) How to perform load management from which operator/plan? (3) How to maintain high availability in distributed system? For example, the system can use the k-safe standard: if the failure of any k servers does not result in any message losses. (4) What are the failure detection and recovery mechanisms? In $Aurora^*$ once upstream node detects some downstream nodes are unavailable, it will search its catalog to find alternative participants to join and continue its query plan.

**Chinese Wall and Cloud Computing:** Wu et al. [73] show how the Chinese Wall policy can be used for information flow control in cloud computing. The authors enforce the policies at the Infrastructure-as-a-Service layer. The authors developed a prototype to demonstrate the feasibility of their approach. In our current work, we have adapted the Chinese Wall policy and demonstrated how stream data generated from the various organizations can be processed in a secure manner. Our work is addressed at the Software-as-a-Service level. Tsai et al. [71] discusses how the Chinese Wall policy can be used to prevent competing organizations virtual machines to be placed on the same physical machine. Graph coloring is used for allocating virtual machines to physical machines such that the Chinese Wall policies are satisfied and better utilization of cloud resources is achieved. Jaeger et al. [43] argue that covert channels are inevitable and propose the notion of risk information flows that captures both overt and covert flows across two security levels. Capturing both covert flows and overt flows in a unified framework allows one to reason about the risks associated with information leakage.

# Chapter 3

# Background: STREAM DSMS

Our choice of MLS-DSMS development is based on the Stanford STREAM Project [8] because of the following reasons. First, the Continuous Query Language (CQL) [9] is well-defined as the semantic foundation for continuous queries. Second, in order to handle queries on different input sources, they provide methods to synchronize timestamps among different input data [62]. Third, the open source DSMS prototype is able to handle basic CQL queries from different clients with multiple server instances.

In this chapter, we present a detailed introduction on Stanford STREAM system. After descriptions of its architecture, query process, and interaction between commands and components, STREAM limitations on secure stream processing are discussed.

## 3.1 STREAM DSMS Overview

Stanford STREAM is referred as *vanilla* DSMS system in this dissertation. The vanilla DSMS is a comprehensive interactive interface for STREAM users, system administrators, and system developers to visualize and modify query plans as well as query-specific and system-wide resource allocation while the system is in operation [69]. We first present the architecture of STREAM then discuss each component of the vanilla DSMS shown in Figure 3.1.

The *server* operates in two phases. In the first phase it registers queries, streams, and relations from the *client* via the command unit. In the second phase, it executes the registered queries and

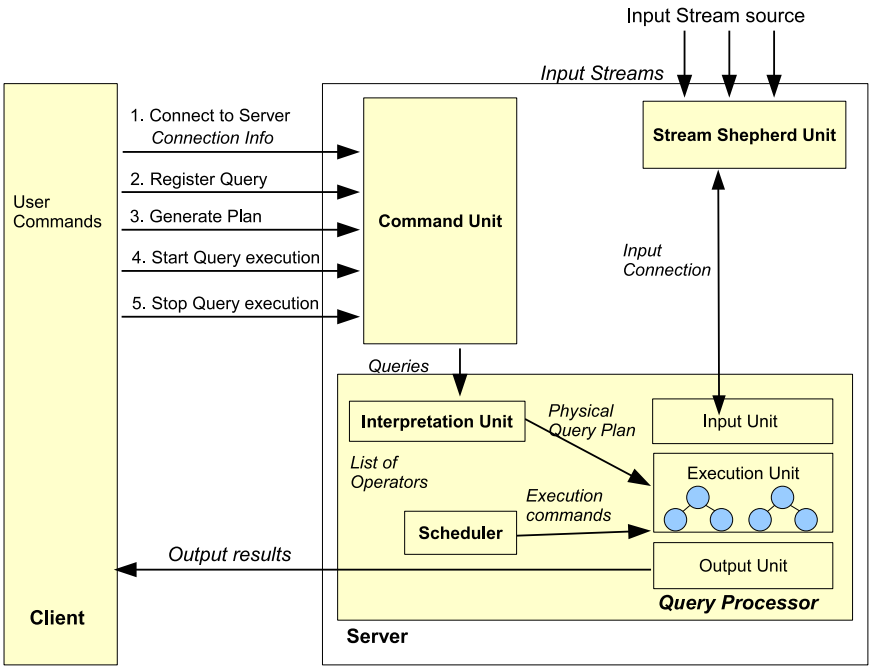


Figure 3.1: DSMS System Architecture

propagates the outputs. No new queries, streams, or relations can be registered Once the second phase starts. The client communicates with the DSMS server via a set of predefined messages in multiple steps. The first two user commands corresponds to the first phase and the next three corresponds to the second phase.

1. *Connect to Server:* The client establishes command communication with the server. The server creates a new server instance specific for that client. All the following command messages are sent to this instance. This does not allow sharing of input streams or queries among different clients. There is no notion of users, authentication, or security levels.

2. *Register Query:* The client registers input stream schemas, relations, and queries. At this stage, a query registration message is sent to the interpretation unit which translates the interpreted query to a logical query plan (a link of operators). The naive physical plan is also generated.

The input streams are connected to the query processor input unit by the stream shepherd unit. Users are required to bind input data sources with the stream schemas explicitly. This is not suitable in the secure DSMS architecture as the users can only access authorized tuples. Thus, we have to modify the registration process. In the output unit, output connection between DSMS and client is established after the queries are registered.

3. *Generate Plan:* Once the DSMS receives command from the client indicating that all queries have been registered and binding of input streams have been completed, it optimizes the naive physical query plans created in the previous step. Also, graphs of physical plans are generated for user view. The generated physical plans are instantiated in the execution unit of the query processor and the list of operators are sent to the scheduler.

   In the replicated MLS-DSMS, the query plans have to be generated in appropriate query processor and should be linked to appropriate single level input streams so that there is no illegal information flow.

4. *Start and Stop Query Execution:* Once the start query execution command is issued, the scheduler instructs the execution unit to start running the specified operators. Input, output, and execution units process stream tuples continuously, and the computation results are sent to the client until user issues the stop query execution command or there are no more input tuples.

## 3.2 Query Process

Let us take a closer look on query process in vanilla system. In general, raw queries received by the server will be processed in the following chain:

```
CQL query -> Syntactic nodes in parse tree -> Semantic objects
-> Logical operation (logical plan) tree
-> Physical operation (physical plan) tree -> Execution units
```

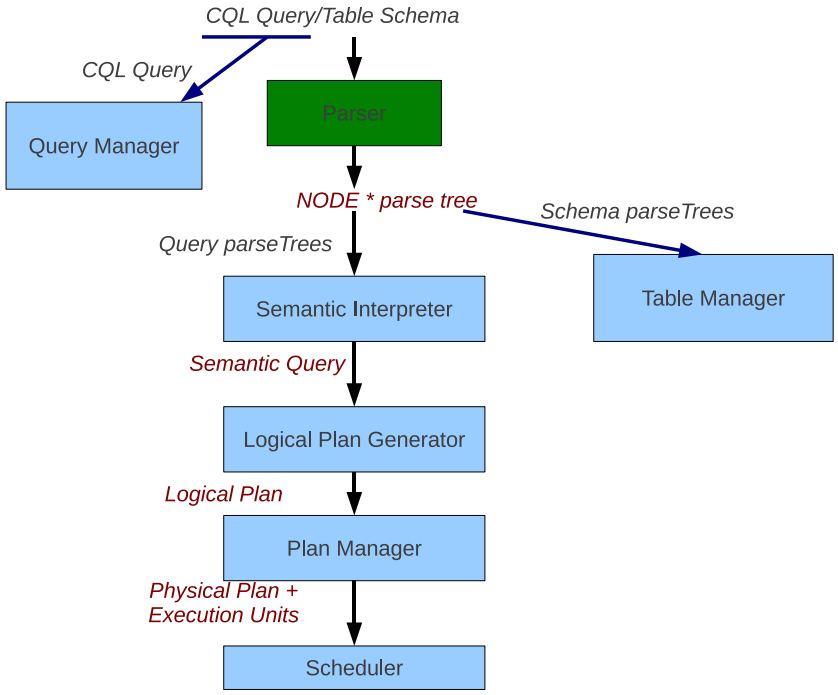


Figure 3.2: Query Process

The system components involved in query process is showed in Figure 3.2. And we explain each component in details one by one. Some descriptions are from STREAM-0.6.0 manual [70].

- Parser. It takes in input stream/relation/view schema and query/monitor specification then decompose the string sentence into different nodes. All parsed info are saved in a *parseTree*. Other process units like query manager can get the registered info from specific nodes.

- Table Manager. In DSMS tables refer to streams and relations in DSMS. The table manager stores the names and attributes of registered stream/relation/view. The input info comes from *parseTree*.

- Query Manager. Registered queries and sub-queries are stored in query manager. Each query will be assigned with a unique identifier (query-id) for further use. [70].

- Semantic Interpreter. Parse tree nodes generated by parser will be handled in this module. The syntactic parse tree is converted to specific a set of semantic objects [70].

- Logical Plan Generator. It transforms an interpreted semantic query to a logical query plan. Those interpreted CQL queries are constructed in certain patterns. The generator checks them by applying set of transformation rules.

- Plan Manager. Physical level entities including operators, queues, synopses,query plans are managed by this module. By reading the logical plan, plan manager instantiates the corresponding physical entities. [70].

- Scheduler. It creates a queue for all operators from physical plans. When scheduler starts running, operators will run one by one and query by query.

From the process tree, there are several products during the query transformation.

- NODEs. ParserCommand handles registered table as well as CQL queries by creating various kinds of nodes as output. Some nodes can be constructed by basic nodes, for example, $REL\_SPEC$ (relation specification) node has two fields $rel\_name$ and $attr\_list$. $attr\_list$ refers to $LIST$ node contains a list of attribute specification nodes $ATTR\_SPEC$. The parsing result is called *parse tree*, which is a node contains set of basic nodes in its fields.

- Semantic query. This object is created by interpreter. This internal representation differs from the parse tree (NODE *) produced by the parser in the following ways: First, input

relation/streams as well as their alias are connected to their internal identifiers. They are assigned with a variable-id, and each attribute belongs to the schema is also assigned with unique attribute-id. So each attribute can be denoted as $< variable-id, attri-id >$. While in parse trees some attributes are implicit. Second, every input stream in the FROM clause is associated with a window. In NODE * there could be streams without a window. The system will add the default UNBOUNDED window for every stream in the FROM clause without a window [70].

- Logical plan. Logical plan is a linked list of logical operators. Operators in plans are constructed as a tree structure, where the operators in lower level provide output their results to the ones in higher level. The bottom operators consume the input stream/relation while the top (root) one produces the final query output. The tree is represented as a linked list where the root operator will be the returned logical/physical plan. Operator is an interface in DSMS, which can be used to represent logical plan, physical plan, and actual executing operators.

  Logical plan operators are different from the the one used in physical plan. In logical plan generator, semantic query is classified as two kinds: Select-From-Where (SFW) and Binary-Join. The generator first produces a naive plan where semantic query is translated with a set of logical operators represent the necessary operations in the query. The execution order of operators is determined during the translation, and it is saved as a linked list. To be specific, each operator contains fields *input* and *output* which denote the previous and next operators in tree structure respectively. After the linked list is created, the top operator (which is the last operator in query) is returned as logical plan object.

In order to show how to construct the linked list in certain order, we take the naive SFW logical plan generation for example. There are several steps of extracting the info from semantic query to create a linked list of operators: 1) Generate a plan that joins the FROM clause tables; 2) Apply WHERE clause predicates over the join; 3) Apply Aggregations and perform group by if necessary; 4) Perform projections specified in SELECT clause; 5) Apply Distinct operator if needed; 6) Apply Relation-to-Stream operators if present.

After a naive plan is generated, logical plan generator performs optimization to remove redundant operators according to the CQL definition. The naive plan is "transformed" to a better logical one after certain optimizations.

- Physical plan. Similar to logic plan, physical plan is a linked list of physical operators. Since the logical plan is the pointer of the top operator, the plan manager will go recursively in the linked list then produce the corresponding physical operators by mapping with logical ones to them as well as additional optimizations. For example, if a logical plan has two sequential select operators, only the child (lower level in tree structure) operator will be kept in physical plan by appending parent's predicate attribute with its own.

  When the physical plan is done, plan manager will generate a special operator *query_source* as the first but artificial operator in the physical plan. This query source operator can be shared among queries requesting the same input. This is a dummy operator found only at the metadata level. [70].

  Another case is that whether the registered query needs to output. If yes, we need to create a specific operator *output* interfaces outside the system. this special operator is also put on the top of the physical plan.

- Execution units. When all queries are registered, plan manager will add all the auxiliary structures like synopses, stores, queues to the plan. These structures as well as related operators will sent to scheduler for execution.

## 3.3 Interactions

In this section, we take a second look at how the interaction commands cooperate with DSMS components. In Figure 3.3, the left side are the public interaction methods and the right side are those important process units in DSMS.

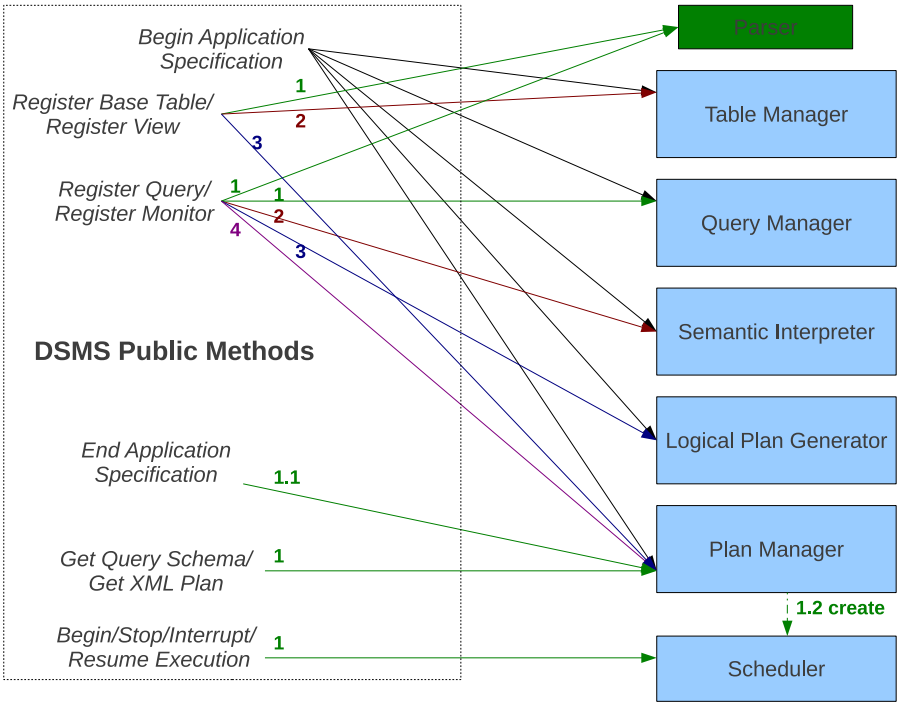


Figure 3.3: DSMS Methods

Public methods will be called by command connection process unit where the DSMS instance is created. When command unit gets specific commands from client, it will call the responding methods in DSMS. Now we explain how these methods interact with process units in DSMS.

1. Begin Application. Five process units except scheduler are initialized. Parser is a process unit which does not need initialization. All the process units except parser is locally initialized in a DSMS instance.

2. Register Base Table/View. Both methods first pass the table name and schema to the parser for node decomposition. Then the table manager will store the name and schema from the parse tree; a table Id is returned. In the third step, base table registration will inform the plan manager about the new table while view registration will indicate the mapping between the previous registered query with a view table Id to the plan manager.

3. Register Query/Monitor. Monitor is a special query reflects the real-time data during query execution.The CQL query will be sent to parser for decomposition and query manager for storage. Parser returns a parse tree and query manager returns a unique query Id. The semantic interpreter transform the nodes in parse tree to a internal query, which would be used in logical plan generator later. After a logical plan is returned, the plan manager will product a corresponding physical plan and bind it with the query Id.

4. End Application. It is called when the client finishes all table/query registration. There are several steps must be done before moving to execution.

   (a) Plan optimization. The non-operation query sources are first removed from physical plan. Then all operators without an output will be added with a *sink* operator in the top. The sink operator consumes all input child operators without an output. After that, the plan manager will try to merge select operations if parent operator is select and the child is select or join. The parent operator will be deleted from plan after the predicates is properly appended to its child operator. Notice that if more than one plan

uses the parent node, the merge cannot be done. Also the manager will try to merge some project operators where parent is project and child is join.

(b) Adding auxiliary (non-operator) structures. These structures like synopses, stores, queues should be attached to each operator in the plan. There are structures need to be added which include the extra *aggregation attribute* in aggregation operator.

Each synopsis is a distinct logical symbol indicates the input/output of an operator while the store is synopsis allocator which allows sharing between synopses. Plan manager will first add the proper synopsis type according to the operator then create stores for synopses. Notice that synopsis-store assignment handles the "synopsis-sharing" (it is actually "store-sharing") between operators. For each operator *o*, manager first checks its output to see if any of the parent operators above *o* have synopses which require that *o* assigns memory for their tuples. If yes, *o* will be attached with a sharable store which will be assigned to the parents' synopses later. Otherwise obsolete stores are created. Each store keeps a stub record indicates synopses are assigned to it. In our MLS development, we need to consider security level on synopsis sharing.

Queues are also attached to each operator. There are three types of queues: *simple queue, writer, and reader*. A simple queue has one source and one destination operators, while a shared queue reader/writer has one source and many destination operators. Writer is the output queue while reader is the input queue for each operator.

(c) Instantiation. Plan manager instantiates memory manager and allocates static tuples contain the constants that are used in operator computations. Execution operator specification will be added to corresponding physical operators in physical plan. After that, the queues will be instantiated and attached to related those execution operators. Then

the plan manager will link the synopses to their stores as well as link the operators to their input stores. The link here means create the implementation of stores and synopses from previous specification.

(d) Scheduler initialization. The execution attribute of operators in physical plan are added into the scheduler.

5. Get Query Schema/XML Plan. Both methods are functions of the plan manager. Get query schema can be returned by a given queryId before or after end of specification for checking the schema definition. XML plan returns the plan for whole queries after end of specification.

   XML plan can be submitted by users to simplify the input/stream registration. A simple example for XML registration plan is presented as following. In the XML file user can specify the schema of input stream, query detail, as well as the construction ways of input data (with/without timestamp, input as a loop).

```
<Script>
  <Table>
    <Name>R</Name>
    <isStream>true</isStream>
    <isBase>true</isBase>
  <Attr>
    <Name>name</Name>
    <Type>2</Type>
    <Len>20</Len>
  </Attr>
  </Table>
  <Query>
    <QueryString>select * from R;</QueryString>
    <isNamed>false</isNamed>
    <hasOutput>true</hasOutput>
  </Query>
  <DemoBinding>
```

```
    <TableName>R</TableName>
    <FileName>R.load</FileName>
    <bLoop>true</bLoop>
    <bAppTs>false</bAppTs>
  </DemoBinding>
</Script>
```

6. Begin/Stop/Interrupt/Assume Execution. All methods refer to the scheduler. If the plan is allowed to execute, scheduler will run the execution operators continuously until interruption/stop commands are received.

## 3.4 Continuous Query Language

In STREAM, the input data can be streams or relations, and a mixture of both. A stream $S$ is a bag of elements $< s, t >$ where $s$ is a tuple belonging to the schema of $S$ and $t$ is the input arrival timestamp of $s$. A relation $R$ is a mapping from each time instant to a finite bag of tuples belonging to the schema of $R$.

Besides, queries are expressed by Continuous Query Language (CQL) which is an extension of Structured Query Language (SQL) by supporting continuous queries on long-running input stream data. In CQL, queries are processed periodically at each time instant *heartbeat* $\tau$, using the input data with a timestamp $t$ where $t \leq \tau$. The input data are partitioned and buffered in a *synopsis* (Syn) with a predefined finite size. At every heartbeat $\tau$, the system processes the query with data set in the window then transfer the results to users continuously. In general, there are three steps during query processing showed in Figure 3.4: (1) Stream-to-Relation (*S2R*): For a heartbeat $\tau$, input stream data with timestamps $t \leq \tau$ are transformed to a temporal relation. (2) Relation-to-Relation (*R2R*): the input "relations" are processed by CQL operators and a resulting relation $R(\tau)$

is generated for each $\tau$. (3) Relation-to-Stream (*R2S*): the resulting $R(\tau)$ is transformed back to stream as the long-running output results.

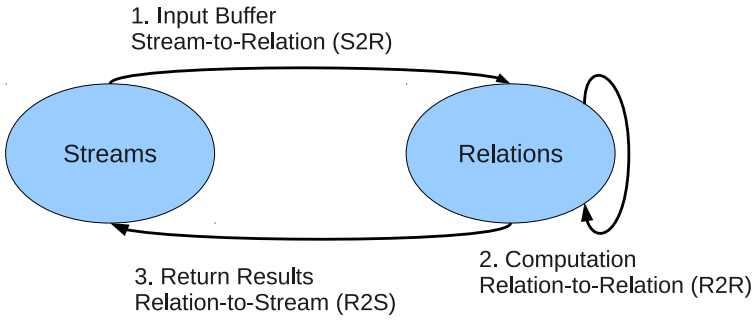


Figure 3.4: Three Steps of Query Processing

Correspondingly, there are three kinds of operators S2R, R2R, R2S to complete the query processing. S2R operators, usually called Sequential Window (Seq-Win), handles query input by generating *slide windows*. The sliding windows are buffering synopses contain a historical snapshot of a finite portion of the stream at each heartbeat $\tau$. Based on different input requests, slide windows are classified as *tuple-based*, *time-based*, and *partitioned-by*. Tuple-based window over Stream $S$ usually uses the form S[Rows N], which requires DSMS to buffer last N tuples of an ordered stream with largest timestamp $t \leq \tau$. Time-based window in the form S[Range T] contains all tuples from timestamp $\tau - T$ to $\tau$. Note that a size constraint cannot be applied to time-based window because of the unpredictable input tuple rates. Partitioned-by window is represented as S[Partitioned By $A_1$, ... $A_k$ Rows N] where $A_1, ..., A_k$ is a subset of attributes defined in stream schema. This window logically partitions $S$ into different substreams based on equality of $A_1$, ...,

$A_k$, and computes a tuple-based window with size $N$ independently on each substream, then take the union of these windows to produce the input relation.

On the other hand, R2R operators are derived from traditional DBMS like selection, projection, join, and aggregations such as maximum, minimum, average, sum, and count. They are responsible for query computation on the mixture of input relations and streams in sliding windows. In step 3, the final results are transformed back to the stream by relation-to-stream R2S operators.

With CQL operators, the query plan can be generated like a SQL query tree where nodes represent the processing units – CQL operators, and the connection edges between nodes are *queues* storing the intermediate or final outputs generated by operators. *Synopsis* is maintained with operators which stores the temporal results for further computation. Except non-blocking operators like selection and projection, only blocking operators like aggregation and join need synopses. Synopsis has different types according to the input window specifications. Tuples buffered in synopsis contain expiration tags calculated from arrival timestamp plus the window size. In query plan, query processing starts from bottom input operators through the output in the top.

The first step of our work is to simulate a battlefield monitoring application. Each soldier equips with sensors sending out Vitals and Positions info to the control center with DSMS continuously. Users like commanders and medics issue real-time queries requesting real-time data analysis based on the infinite remote streaming data. The schemes are showed below:

```
Vitals(soldier id (sid), blood pressure (bp), pulse rate (pr),
       weight (weight));
Positions(soldier id (sid), latitude (lat), longitude (lon));
```

Timestamp *ts* is attached to each tuple by vanilla DSMS indicating the arrival time instant. The system forms two streams Vitals and Positions collecting information from all soldiers, com-

putes query request in real-time, and transmits results back to users. CQL queries $Q_a$, $Q_b$ and $Q_c$ illustrates the usages of three kinds of windows in vanilla system.

```
Qa:  Find soldiers' Vital and Position information from last
     100 input tuples where the soldier is in longitude "12E".
CQL: SELECT *  FROM Vitals[Rows 100] V, Positions[Rows 100] P
     WHERE V.sid = P.sid AND P.lon = "12E"

Qb:  Compute average of soldiers' blood pressure from the data
     received in 5 minutes.
CQL: SELECT AVG(bp)
     FROM Vitals[Range 5 Minutes];

Qc:  Compute the number of soldiers located in different
     latitudes from last 500 input tuples.
CQL: SELECT COUNT(sid)
     FROM Vitals[Partition By lat Rows 500];
```

## 3.5 Limitations

With the architecture design and query processing STREAM vanilla DSMS, four of five the missing stream processing requirements [63] in traditional DBMS are satisfied. (1) STREAM is an active management system, which keeps stream data moving without storing the whole first; (2) CQL is proposed to specify stream-specific queries; (3) The continuous queries can be handled so that output results are predictable and repeatable; (4) Query inputs can be live stream or tables in finite size.

The requirement on handling missing/out-of-order/delayed inputs are not considered in STREAM because integration of those extra mechanisms causes high latency. Besides, STREAM prototype does not focus on the correct arrival orders of inputs such as stock market analysis. In fact, a possible solution on delayed/out-of-order of input streams is proposed in Aurora DSMS using *slack*

*method* in [1]: before query processing, the system uses 2 passes bubble sort on a limit size of inputs trying to provide better ordered input sequence. The sorting method uses small buffers and computation power not causing big overhead in DSMS.

Our work is applying DSMS to multilevel scenario applications: input data are sensitive and queries from different users are classified. In this section, we discuss the limitations of vanilla STREAM in in two main aspects: security preservation and performance improvement.

## Security Threats

In our design, security level is a special attribute existed in all input tuple schemes. Each input should be attached with one security level then accessibility is determined. Queries with lower level cannot access inputs with higher level. An *authentication* module during client-server connection must be added to the original DSMS.

To ensure security preservation, some might argue that adding a filtering in query pre-processing might solve this problem. However, the *inference problem* exists in STREAM system. For example, we have two queries issued by a low level (L) user:

```
Q1: Return max value and id of every two new inputs.
CQL: SELECT id, MAX(value) FROM Input[Rows 2]
     WHERE level = "L";

Q2: Return all ids and values of every two inputs.
CQL: SELECT id, value FROM Input[Rows 2]
     WHERE level = "L";
```

Suppose there are three inputs with schema (level,id,value) arrives in the following order: (L,First,100),(L,Second,90),(H,Third,150). Since this is a low level query, only the first two inputs will be considered for maximum computation. The output result for $Q_1$ is 100,100,90, the

number of computation times is 3; While the output result for $Q_2$ is (First,100), (Second,90), the computation times is 2. The reason is that the arrival of (H,Third,150) **expires** the first input (L,First,100), which causes the computation of maximum. By observing those difference between the two queries, low level user is able to infer the existence of inputs from high level.

Another threat is *covert channel problem*. By STREAM DSMS design, it uses a round-robin scheduler to run all registered queries. Under sensitive info applications, malicious users can build up a timing covert channel easily by affecting the response time of low level users. Suppose there are two users $U_h$ and $U_l$ in different level high and low. In timestamp 1, both users issue only 1 identical query $Q$ respectively. $Q$ returns info from a stream with low level inputs. At timestamp 1 response time for $U_l$ should be fast. At timestamp 2, $U_h$ issues 50 $Q$s and $U_l$ maintains same 1 $Q$. The response time in timestamp 2 for $U_l$ will be significantly delayed by observation. By setting up patterns such as issuing different numbers of queries at pre-designed timestamps, $U_h$ builds a channel to send 0/1 messages (suppose 0 for fast and 1 for slow) to $U_l$ periodically. The design of scheduler must be reconsidered under secure stream processing applications.

## Performance Issues

Vanilla DSMS provides input sharing for all registered queries only if they are accessing the same input streams and they are issued by the same user. Stream inputs are not shared between different clients. Besides, query processing storage and results are not shared between queries, even though they are identical in syntax and query plans. In our work we aim to reuse the process and results between queries in certain similar forms.

The expression power of CQL can also be improved by supporting queries in certain levels. For example in $Q_1$, original CQL cannot buffer tuples only in level L at the very beginning. We've

extended such limitations in our MLS models.

In the following chapters, we are providing the MLS models and formalizations. Then in in different MLS-DSMS implementations, we propose solutions to handle the limitations of the vanilla DSMS.

# Chapter 4

# Multilevel Security Formalization

## 4.1 Multilevel Security Model

In order to support multilevel security (MLS), we first need to define MLS model for the system as well as the CQL queries. An MLS-DSMS is associated with a security structure that is a partial order, (**L**, $<$). **L** is a set of security levels, and $<$ is the dominance relation between levels. If $L_1 < L_2$, then $L_2$ is said to strictly dominate $L_1$ and $L_1$ is said to be strictly dominated by $L_2$. If $L_1 = L_2$, then the two levels are said to be equal. $L_1 < L_2$ or $L_1 = L_2$ is denoted by $L_1 \leq L_2$. If $L_1 \leq L_2$, then $L_2$ is said to dominate $L_1$ and $L_1$ is said to be dominated by $L_2$. $L_1$ and $L_2$ are said to be incomparable if neither $L_1 \leq L_2$ nor $L_2 \leq L_1$. We assume the existence of a level U (Unclassified), that corresponds to the level unclassified or public knowledge. The level U is the greatest lower bound of all the levels in **L**. Any data object classified at level U is accessible to all the users of the MLS DSMS. Each MLS DSMS object $x \in \mathbf{D}$ is associated with exactly one security level which we denote as $L(x)$ where $L(x) \in \mathbf{L}$. (The function $L$ maps entities to security levels.) We assume that the security level of an object remains fixed for the entire lifetime of the object.

The users of the system are cleared to the different security levels. We denote the security clearance of user $U_i$ by $L(U_i)$. Consider a military setting consisting of four security levels: Top Secret (TS), Secret (S), Confidential (C) and Unclassified (U). Their dominating relation is U $<$ C $<$ S $<$ TS. The user Jane Doe has the security clearance of Top Secret, $L(JaneDoe)$ = TS.

Each user has one or more associated principals. The number of principals associated with the user depends on their security clearance; it equals to the number of levels dominated by the user's clearance. In our example Jane Doe has four principals: JaneDoe.TS, JaneDoe.S, JaneDoe.C and JaneDoe.U. During each session, the user logs in as one of the principals. All processes that the user initiates in that session inherit security level of the corresponding principal.

Each continuous query $Q_i$ is associated with exactly one security level. The level of the query remains fixed for the entire execution. The security level of the query is the level of the principal who has submitted the query. For example, if Jane Doe logs in as JaneDoe.S, all queries initiated by Jane Doe during that session will have the level Secret (S). A query consists of one or more operators $OP_i$. We require a query $Q_i$ to obey the simple security property and the restricted $\star$-property of the Bell-Lapadula model [16]. In MLS-DSMS, subjects are active elements of the system like operators execute queries, while objects are passive elements of the system that contain information such as queues, tuples, and input streams.

- A subject $S_i$ with $L(S_i) = C$ can read an object $x$ only if $L(x) \leq C$.
- A subject $S_i$ with $L(S_i) = C$ can write an object $x$ only if $L(x) = C$.

Let us consider the benefits of applying BLP model in multilevel security. Simple security property ensures no read-up and the restricted $\star$-property allows only write-equally (no write-down or write-up). Both ensure information can only flow up from low to high level.

An MLS-DSMS deals with different types of data objects. In real-time stream monitoring applications, we have sensors that capture information. We refer to these sensors as *stream sources*. One or more sensors can add tuples to a stream. Each sensor or stream source $SS_i$ is associated with a security level $L(SS_i)$. The location of the sensor determines its security level. For example,

if a sensor $SS_i$ is located in a top-secret location, then its security level $L(SS_i)$ = TS. We refer to the data stream generated from the sensors as the source data stream. The source data stream $S_i$ generated from the sensor $SS_i$ inherits the security level of the sensor, that is, $L(S_i) = L(SS_i)$. All the sensors writing to a stream have the same security level as the stream. Each tuple $t$ belonging to the source data stream inherits the security level of the stream source, that is, $L(t) = L(SS_i)$. All the tuples in the source data stream are at the same level. We assume that each source data stream $SS_i$ is associated with a single security level. This assumption is required to satisfy the restricted $\star$-property of the BLP model. Though, we assume source streams are single level, we consider our stream inside the data stream management system (i.e., once the tuples enter the system) to be multilevel.

For each input data record, multilevel security can be supported at two *granularities*: tuple and attribute. In this dissertation, we put security enforcement at tuple by appending level attribute to all input data.

## 4.2 Multilevel Queries

Consider the battlefield monitoring situation again and now each input tuple comes with security labels. Suppose control center runs a vanilla DSMS handling four queries ($Q_d, Q_e, Q_f$ and $Q_g$) using the CQL language that does not support MLS specifications. So we have added the MLS clauses, appropriately.

```
Qd:  Compute the average bp and pr from the last 100 input data
     in the unclassified level
CQL: SELECT AVG(bp), AVG(pr) FROM Vitals[Rows 100]
     Where level = U
```

In query $Q_d$, the stream window maintains the last 100 tuples in the synopses. This particular

Table 4.1: User-CQ Relationship

| USER | QUERY LEVEL | QUERY | RESPONSE REQUESTED |
|---|---|---|---|
| Bob | TS | $Q_d$ | at level U |
| Kim | S | $Q_e$ | all levels dominated by S |
| Jim | U | $Q_f$ | all levels dominated by U |
| Alice | TS | $Q_g$ | all levels dominated by TS |

query can be executed by a user at any level, since the security level is unclassified (U). Moreover, the results returned by the query is independent of the security level at which the query is issued. We term queries that run at a particular level as **single level** queries.

Consider queries $Q_e, Q_f$, and $Q_g$ where the same query is issued by users logged on at different security levels as shown in Table 4.1. Note that, there is no explicit mention of security level in this query. Thus, queries $Q_e, Q_f$, and $Q_g$ are syntactically equivalent but will return different results. Queries $Q_d$ and $Q_f$ are syntactically different but will return the same response. We term queries like $Q_e$ and $Q_g$ as **multiple level** queries as the result combines multiple levels.

```
Qe, Qf & Qg: Select AVG(bp),AVG(pr) From Vitals[Rows 100];
```

Consider a variation of the queries $Q_e$, $Q_f$, and $Q_g$ using the partitioned clause shown below as $Q'_e, Q'_f$, and $Q'_g$. The results for $Q'_e, Q'_f$, and $Q'_g$ will be **partitioned** by the security levels. For example, query $Q'_g$ will produce averages for each level separately rather than the **combined** average for every 100 tuples across all levels.

```
Q'e, Q'f & Q'g: Select AVG(bp),AVG(pr) From Vitals[Rows 100
                Partitioned By level];
```

MLS DSMS adapts CQL so that various MLS continuous queries (single level, multiple level, partitioned, combined) can be expressed in the framework. In this research, the new language called **MLS-CQL** will extend the CQL syntax and define a new semantics that will dictate how

MLS queries are interpreted and processed and what results can safely be returned to the user. The formalism should capture the security level of the user issuing the query using the notion of dominance of security levels. The formalism will also allow us to reason about query equivalences, which, in turn, will help with query rewriting and query optimization. Besides, the formalization is critical as it directs the design of the architecture and will also help in proving the correctness and soundness of the algorithms.

## 4.3 Stream-to-Stream Window Operator

In CQL, condition filtering can only be specified in WHERE clause via select operator. S2R Window operators are just used for buffering, which limits the flexibility of continuous queries under multilevel secure circumstance: user might only need to buffer data within specific levels. On the other hand, lack of control on buffering data in window operator might cause security threats. For example, an U level user issues two queries at the same on input stream Vitals:

```
Q1: SELECT AVG(bp) FROM Vitals[ROWS 3]
    WHERE level = U;
Q2: SELECT bp FROM Vitals[ROWS 3]
    WHERE level = U;
```

Suppose there are 4 input tuples(timestamp,level,bp) arrive in the timestamp order, the computation results are performed on data in level U, which is showed in Table 4.2.

The U level user knows all inputs in timestamp 1, 2, and 4 except 3. From the computation result in timestamp 4 turns out to be 130 instead of 120, he will know there is some high level input arrive among timestamp 3 and 4, even though he does not see the input from the queries. As a result, the existence of sensitive information is leak to unauthorized users.

Table 4.2: Information Leak Example

| TIMESTAMP | INPUT | WINDOW BUFFER | RESULT |
|---|---|---|---|
| 1 | (1,U,100) | (1,U,100) | 100 |
| 2 | (2,U,100) | (1,U,100),(2,U,100) | 100 |
| 3 | (3,TS,140) | (1,U,100),(2,U,100) (3,TS,140) | N/A |
| 4 | (4,U,160) | (2,U,100),(3,TS,140) (4,U,160) | **130** (*NOT 120*) |

To prevent information leak as well as provide flexible continuous query in MLS, We propose to introduce secure *stream-to-stream* window operators that will provide a filter based on the security level. Not like CQL traditional window operator converts a stream to a relation, our extension provides a filtering operation on the stream prior to the application of the window operator. The syntax is straightforward: users can request data in a special level set $L$ like "$level$ in $L$" inside window specification. Suppose a TS level user issues $Q_x$:

```
Qx: Compute the average blood pressure (bp) of the soldiers from
    last 100 input data where the clearance is in level C or U
CQL: Select AVG(bp) From Vitals[Rows 100 level in {C,U}];
```

The window operator is now stream-to-stream enabled. It should only buffer 100 input tuples in level C or U. Data in other levels like TS and S are discarded even though they are accessible by the TS user. Comparatively, the system can conduct an authorization check in window level specification. If a U level user issues the same $Q_x$, the system should reject it since the query tries to access data in unauthorized level C. By such specification the system can identify the acceptable levels of the raw input data.

Level specification can also be done in WHERE clause of select. According to execution semantics in traditional DBMS and CQL, the select operation is applied in where clause after

input data tables are chosen. Similarly in MLS-DSMS, input streaming data are first buffered in window operator, then filtered by select with level specification when tuples exit from the window. Suppose a TS level user issues a query $Q_y$:

```
Qy: Compute the average blood pressure (bp) of the soldiers
whose clearance in level C or U from last 100 input data
CQL: Select AVG(bp) From Vitals[Rows 100] Where level in {C,U};
```

The difference between $Q_x$ and $Q_y$ is the input data used for average computation. In $Q_y$, the window operator buffers data with level in $\{$TS,S,C,U$\}$ and then only those in level $\{$C,U$\}$ will be computed for average. Due to the different level specification ways between window and select operators, the CQL representations as well as the computation results are distinguished.

With the MLS-CQL formalization, we can express the level request during query process step 1 S2R and step 2 R2R via window and select operators respectively. The level specifications help identify illegal queries request on unauthorized data in higher level and perform possible sharing analysis before query execution. In chapters ofreplicated and trusted MLS-DSMS, we will discuss the system architectures to address MLS-CQL queries.

# Chapter 5

# Replicated MLS-DSMS

## 5.1 Multilevel Secure DSMS Architectures

In order to respect MLS constraints, vanilla STREAM architecture should be extended to achieve logical isolation across the security levels. MLS-DBMS architectures [2, 24, 32] such as *partitioned, replicated,* and *trusted* have been investigated to see if these ideas can be applied to DSMSs. From those researches the critical issue is to make a decision for each component as to whether it should be **trusted** or not. Trusted components can handle tuples at different security levels without causing illegal information flow. Each untrusted component has a security level associated with it and can be uniquely defined or replicated for handle queries in different levels. In general, different architecture choices affect the design of components as following:

- In partitioned architecture, each component receives only those tuples that have the same security level as the component. The number of components of specific function is identical to the number of security levels.

- If we use the replicated architecture, some components like query processors can receive tuples that are dominated by the level of that component. So the inputs in lower level are duplicated to components in the same kind (e.g., query processor) but in different levels.

- In trusted architecture, all system components are unique and assigned with highest security level. In other words, each component handles query request from all different levels.

Now we need to figure out which architecture is suitable for MLS query processing. We begin with discussing the necessary components for MLS-DSMS developments. First, the **user interface** (UI) component is needed which consists of the input processor and output manager. We can make this component trusted and allow it to accept queries and send results to users at different security levels. The UI MLS extension is identical to the three proposed architectures.

We next discuss the **input stream handling** component. Recall that sensors write tuples to source data streams that have the same security level adhering to the restricted *-property. Often times the source data streams having the same schema are merged into a single multilevel stream. The stream shepherd operator is the entry point for an input stream and it also acts as the clearing house. It converts tuples to internal representation, and writes to appropriate queues. Thus, trusted stream shepherd operator is needed to handle multiple levels. The implementation can be done in the three architectures.

Query processor is the DSMS execution unit runs query plans in different security levels. The design decisions of components such as **operators, queues,** and **synopsis** impact the system performance as well as the security protection. We need to make a choice as to whether these would be partitioned, replicated, or trusted.

Suppose partitioned architecture is used for the synopses. For processing the query shown below, we have two synopsis Syn(L) and Syn(H), where Syn(L) contains only low-level tuples and Syn(H) contains high-level tuples. Consider the following query issued by a high-level user:

```
Select sid, bp From Vitals[Rows 50,000];
```

For processing this query, we have to look at both Syn(H) and Syn(L). Moreover, we need to figure out how many tuples to check for each one as the total tuples in the sliding window is 50,000 according to the arrival timestamps. In order to provide output in correct timestamp order,

the result combination and reconstruction from low and high servers in partitioned architecture can cause high latency.

On the other hand, in replicated synopsis we will have two levels of synopsis Syn(L) and Syn(H) while in trusted synopsis only Syn(H) is needed. Since Syn(H) will store all the tuples, the above query is more easily answered. The only negative side for replicated architecture is that the low-level tuples are stored twice, costing space. For real-time queries request mixed streaming data, query processing in replicated and trusted architectures appears to be much faster.

We next look at the property of query processor. In trusted architecture, we can have unified trusted query processor that handles queries at all security levels. Alternatively, we may have query processors at each security level that are responsible for the queries at the corresponding security levels in replicated architecture. In addition to the above components, the designs of scheduler as well as load shedding mechanism are also critical in performance and security issues.

By the analysis above, our research focus on centralized architectures as replicated and trusted rather than the partitioned design. Our choice is based on the two reasons: first, replicated and trusted architectures provide better performances on mixed level data processing, which is often happened in MLS applications. Second, the two centralized designs face challenges on illegal information flow threats on query scheduling and executions, as well as sharing common resources between processes across different levels. The explorations of security and performance topics will benefit researches on MLS applications, DBMS development, and DSMS with security enforcements. In this and next sections, we will explain the replicated and trusted structure designs in details.

## 5.2 Replicated MLS-DSMS Architecture

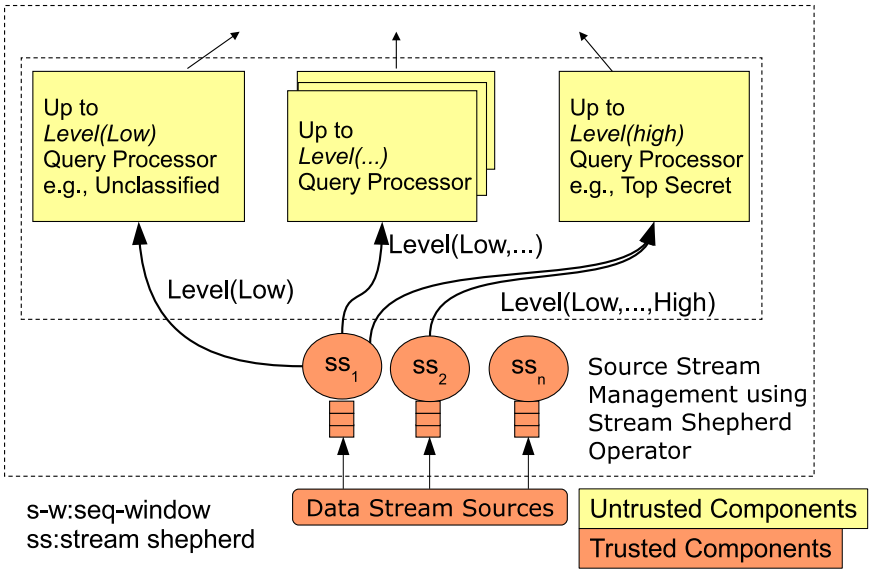


Figure 5.1: Replicated MLS-DSMS Architecture

In the following sections, we would like to present the overview of replicated MLS-DSMS. Then we provide details on how MLS queries are shared, generated, and executed. After that, scheduling methods in this system are discussed.

The overview of replicated architecture is shown in Figure 5.1. It supports single and multiple level queries. The replicated architecture has multiple query processors that execute queries dominated by a particular level. The stream shepherd operators enqueue all tuples to a processor where inputs' level is up to the level of the processor. For instance, all the tuples up to level "high" are enqueued to Level(high) processor.

Let us consider queries $Q_e, Q_f$ and $Q_g$ in Table 4.1 executed by replicated MLS DSMS again. Query $Q_f$ is executed in the processor with Level(U); $Q_g$ and $Q'_g$ are executed in the processor with Level(TS). Though $Q_e$ is syntactically equivalent to $Q_g$ they are not shared as they are executed by

different processors.

```
Qe, Qf & Qg: Select AVG(bp),AVG(pr) From Vitals[Rows 100];
Q'e, Q'f & Q'g: Select AVG(bp),AVG(pr) From Vitals[Rows 100
                Partitioned By level];
Kim in level S issues Qe;
Jim in level U issues Qf;
Alice in level TS issues Qg;
```

## 5.3 Shared Query Processing

In this section, we give examples of MLS CQL queries and discuss how the processing of such queries can be shared.

### 5.3.1 MLS-CQL Queries

We have an additional attribute called *level* in each schema of a stream or relation. We can query this attribute, or submit queries based on this attribute.

An MLS-CQL query can include the LEVEL attribute in the WHERE clause, SELECT clause, and window specification. Let us consider the following examples, based on data streams Vitals and Positions.

```
SELECT AVG(bp) WHERE LEVEL = "S" FROM Vitals [ROWS 100]

SELECT AVG(bp) FROM Vitals [ROWS 100 LEVEL = "S"]

SELECT AVG(bp) FROM Vitals [ROWS 100] WHERE LEVEL = "S"
```

In the first query the WHERE clause conditions are applied before a tuple enters a window. In the second query, the window keeps only tuples based on the condition specified. In the third query, the window maintains 100 tuples, but the WHERE clause is applied during AVG calculation. The first and second queries are equivalent. Note that from previous section MLS-CQL Formalization,

we use the second query to classify level filtering between select and window operators. Our MLS DSMSs are able to address all three types of queries.

An MLS-CQL query may not reference the security level attribute at all. The query below demonstrates this – it joins tuples from two streams. The sliding windows maintain the last 100 tuples for computations.

```
SELECT AVG(bp), AVG(pr)
FROM Vitals[ROWS 100], Position[ROWS 100]
WHERE Vitals.sid = Position.sid
```

Table 5.1: Continuous Queries

| **Query** | **User** | **Login Lv.** | **Query Specification** |
|---|---|---|---|
| $Q_1/Q_1'$ | Ann/Bob | $H$ | SELECT AVG(bp)<br>FROM Vitals [PARTITIONED BY LEVEL ROWS 20] |
| $Q_2$ | Carl | $H$ | SELECT AVG(bp) FROM Vitals [ROWS 20]<br>WHERE LEVEL = "L" |
| $Q_3$ | Dan | $H$ | SELECT AVG(bp)<br>FROM Vitals [PARTITIONED BY LEVEL ROWS 5]<br>WHERE bp > 50 |
| $Q_4$ | Dan | $H$ | SELECT AVG(pr)<br>FROM Vitals [ROWS 10] V, Position [ROWS 10] P<br>WHERE V.sid = P.sid AND bp > 120 AND lon = "4E" |
| $Q_5$ | Ellen | $H$ | SELECT V.sid, pr<br>FROM Vitals [ROWS 10] V, Position [ROWS 10] P<br>WHERE V.sid = P.sid AND bp > 120 AND lon ="4E" |
| $Q_6$ | Frank | $H$ | SELECT sid, bp FROM Vitals<br>WHERE bp > 120 |
| $Q_7$ | Gail | $H$ | SELECT sid, bp, pr FROM Vitals<br>WHERE LEVEL = "L" AND bp > 120 |
| $Q_8$ | John | $H$ | SELECT sid FROM Vitals<br>WHERE pr > 100 |

Table 5.1 lists eight queries supported by replicated MLS-DSMS. For simplicity, we use only

two levels high(H) and low(L) in those examples. And we consider only two types of windows: tuple-based ($Q_2$, $Q_4$, $Q_5$ from Table 5.1) and partitioned by windows ($Q_1$ and $Q_3$ from Table 5.1) [9].

Processing each MLS query in our architecture involves several steps. First, the selection condition of the query is written in conjunctive normal form. Subsequently, we generate the query plan. In our framework, we represent a query plan in the form of a tree which we refer to as an *operator tree*. Note that, many operator trees may be associated with a query corresponding to the different plans. However, we show just one such tree for each query. The formal definition of an operator tree appears below.

**Definition 1.** *[Operator Tree] An* operator tree *for a query $Q_x$, represented in the form of $OPT(Q_x)$, consists of a set of nodes $N_{Q_x}$ and a set of edges $E_{Q_x}$. Each node $N_i$ corresponds to some operator in the query $Q_x$. Each edge $(i, j)$ in this tree connecting node $N_i$ with node $N_j$ signifies that the output of node $N_i$ is the input to node $N_j$. Each node $N_i$ is labeled with the name of the operator $N_i.op$, its parameters $N_i.parm$, the synopsis $N_i.syn$, and input queues $N_i.inputQueue$ which are used for its computation. The label of node $N_i$ also includes the output produced by the node, denoted by $N_i.outputQueue$, that can be used by other nodes or sent as response to the user.*

Operator trees for queries $Q_6$ and $Q_7$ defined in Table 5.1 appear in Figures 5.2(a) and 5.2(b), respectively. An operator tree has all the information needed for processing the query. Specifically, the labels on the node indicate how the computation is to be done for evaluating that operator, where an operator is the basic using of data processing in a DSMS. The name component specifies the type of the operator, such as, *select*, *project*, *join* and *average*. The parameter indicates the set of conjuncts for the *select* operator, or the set of attributes for the *project* operator. The parameter is

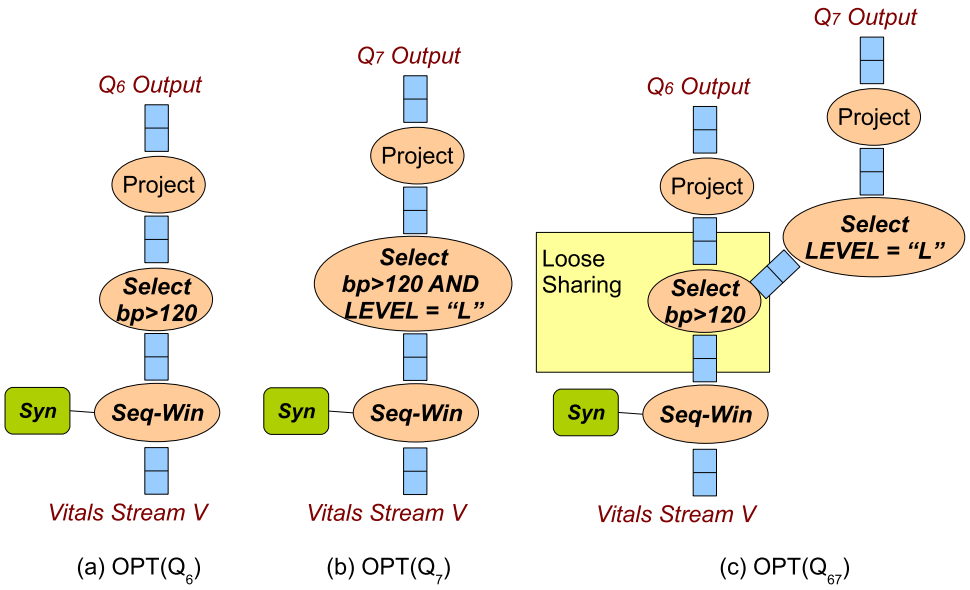


Figure 5.2: Operator Tree for $Q_6$ and $Q_7$

denoted as a set. For the *select* operator, parameter is the set of conjuncts in the selection condition. For the *project* operator it is a set of attributes. The synopsis is needed for the blocking operators, such as, *join* and *aggregate*, and has type (tuple-based or partitioned by) and size as its attributes. The input queues are derived from the streams and relations needed by the operator.

We use the streams (Vitals and Position) and continuous queries shown in Table 5.1 to discuss query processing. We also assume the tuples sent by soldiers involved in a highly classified mission to be classified as high ($H$) and other missions to be classified as low ($L$). Medics or users can login in at different levels and submit queries. Also note that in Table 5.1 all queries are issued in high ($H$) level. The main reason to choose one level is that all queries issued by a user logged in at that level is processed by a query processor running at the that level. Hence we use examples from $H$ level to introduce and discuss various sharing methods. All these queries are executed by one query processor at level high, shown in Figure 5.1.

Queries $Q_1$ and $Q_1'$, issued by Ann and Bob respectively, compute the average blood pressure

of the last 20 tuples at each level in Vitals stream. Query $Q_2$ computes the average blood pressure of the last 20 tuples having level $L$. Query $Q_3$ computes the average blood pressure for the last 5 tuples at each level where the pressure is greater than 50. In queries $Q_4$ and $Q_5$, the last 10 tuples that satisfy the selection conditions are maintained in the synopses and are joined. Average and projection are computed over the results from the *join*. In queries $Q_6$ to $Q_8$, there are only selection conditions and projection (duplicate preserving) operations.

### 5.3.2 Query Sharing

Typically, in a DSMS there can be several queries that are being executed concurrently. Query sharing will increase the efficiency of these queries. Query sharing obviates the need for evaluating the same operator(s) multiple times if different queries need it. In such a case, the operator trees of different queries can be merged. In the Figure 5.3, we show how the operator trees of $Q_4$ and $Q_5$ can be merged. Both use the same *seq-win* operator, as there is one seq-win operator per stream. Later we will formalize how such sharing can be done.

In our replicated MLS-DSMS query processing architecture, we focus on sharing queries to save resources such as CPU cycles and memory usage. In our architecture, we share queries that are submitted by users with the same principal security level as all these queries run in the same query processor. Since queries shared have the same security level, our replicated MLS-DSMS query processor avoids security violations like covert channel during sharing.

We next formalize basic operations that are used for comparing the nodes belonging to different operator trees. Such operations are needed to evaluate whether sharing is possible or not between queries. We begin with the equivalence operator. If nodes belonging to different operator trees are equivalent, then only one node needs to be computed for evaluating the queries corresponding to

these different operator trees.

**Definition 2.** *[Equivalence of Nodes] Node $N_i \in N_{Q_x}$ is said to be* equivalent *to node $N_j \in N_{Q_y}$, denoted by $N_i \equiv N_j$, where $N_i$, $N_j$ are in the operator trees $OPT(Q_x)$, $OPT(Q_y)$ respectively, if the following condition holds: $N_i.op = N_j.op \wedge N_i.parm = N_j.parm \wedge N_i.inputQueue = N_j.inputQueue$.*

In some cases, for evaluating node $N_i$, belonging to operator tree $OPT(Q_x)$, we may be able to reuse the results of evaluating node $N_j$ belonging to operator tree $OPT(Q_y)$. This is possible if the nodes are related by the subsume relationship defined below. Such relationship is possible when the operators match and are non-blocking and the operator parameters are related by a subset relation.

**Definition 3.** *[Subsume Relation of Nodes in Replicated DSMS] Node $N_i \in N_{Q_x}$ is said to be* subsumed *by node $N_j \in N_{Q_y}$, denoted by $N_i \subseteq_R N_j$, where $N_i$, $N_j$ are in the operator trees $OPT(Q_x)$, $OPT(Q_y)$ and are referred to as* subsumed node, subsuming node *respectively, if the following conditions hold:*

1. *Condition 1:*
   - *Case 1 [$N_i.op = project$]:* $N_i.op = N_j.op \wedge N_i.parm \subseteq N_j.parm \wedge N_i.inputQueue = N_j.inputQueue$.
   - *Case 2 [$N_i.op = select$]:* $N_i.op = N_j.op \wedge N_j.parm \subseteq N_i.parm \wedge N_i.inputQueue = N_j.inputQueue$.
2. *Condition 2: $N_i.op$ is a non-blocking operator.*

Consider the *select* nodes of the operator trees of query $Q_6$ and $Q_7$ shown in Figure 5.2, where the *select* node of $Q_7$ is subsumed by the *select* node of $Q_6$.

We have different forms of sharing that are possible in our architecture which we now discuss.

**Complete Sharing**

The best form of sharing is complete sharing where no additional work is needed for a new query. However, in order to have complete sharing, the two queries must have equivalent operator trees. The notion of equivalence of operator trees is given below.

**Definition 4.** *[Equivalence of Operator Trees] Two operator trees $OPT(Q_x)$, $OPT(Q_y)$ are said to be equivalent, denoted by $OPT(Q_x) \equiv OPT(Q_y)$ if the following conditions hold.*

1. *for each node $N_i \in N_{Q_x}$, there exists a node $N_j \in N_{Q_y}$, such that $N_i \equiv N_j$.*

2. *for each node $N_p \in N_{Q_y}$, there exists a node $N_r \in N_{Q_x}$, such that $N_p \equiv N_r$.*

The formal definition of complete sharing appears below.

**Definition 5.** *[Complete Sharing] Query $Q_x$ can be* completely shared *with an ongoing query $Q_y$ submitted by a user at the same security level only if $OPT(Q_i) \equiv OPT(Q_j)$.*

Complete sharing is possible only when the queries are equivalent. For example, queries $Q_1$ and $Q_1'$ have identical operator trees and can be completely shared. In such cases, we do not need to do anything else for processing the new query. However, this may not happen often in practice.

**Partial Sharing**

We next define partial sharing which allows multiple queries to share the processing of one or more nodes, if they are related by the equivalence or subsume relation.

**Definition 6.** *[Partial Sharing] Query $Q_x$ can be* partially shared *with an ongoing query $Q_y$ submitted at the same security level only if the following conditions hold*

1. $OPT(Q_x) \not\equiv OPT(Q_y)$
2. *there exists $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that one of the following holds: $N_i \equiv N_j$, $N_i \subseteq_R N_j$ or $N_j \subseteq_R N_i$.*

We have two forms of partial sharing which we describe below. The main motivation is the sharing of blocking operators have to be handled differently from non-blocking operators. The sharing of blocking is more restrictive in which the conditions for *join*, for example, must exactly match the other query operator. On the other hand, with non-blocking operator they can be subsumed. The formal definition of these two forms of sharing appears below.

**Definition 7.** *[Strict Partial Sharing] Query $Q_x$ can be* strict partially shared *with an ongoing query $Q_y$ submitted at the same security level only if the following conditions hold*

1. $OPT(Q_x) \not\equiv OPT(Q_y)$
2. *there exists $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that $N_i \equiv N_j$*
3. *there does not exist $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that $N_i \subseteq_R N_j$ or $N_j \subseteq_R N_i$.*

**Definition 8.** *[Loose Partial Sharing] Query $Q_x$ can be* loose partially shared *with an ongoing query $Q_y$ submitted at the same security level only if the following conditions hold*

1. $OPT(Q_x) \not\equiv OPT(Q_y)$
2. *there exists $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that $N_i \subseteq_R N_j$.*

In the loose partial sharing, we will have a node on the ongoing query that subsumes a node of an incoming query. When nodes are related by subsume relation, then it is possible to decompose the subsumed nodes. The decomposition tries to make use of operator evaluation of the subsuming node in order to evaluate the subsumed node. The decomposition is formalized below.

**Definition 9.** *[Decomposition of Subsumed Nodes in Replicated DSMS] Let* $N_i \subseteq_R N_j$ *where* $N_i \in OPT(Q_x)$ *and* $N_j \in OPT(Q_y)$*. Node* $N_i$ *can be decomposed into two nodes* $N_i'$ *and* $N_i''$ *in the following manner.*

*Node* $N_i'$

1. $N_i'.op = N_j.op$
2. $N_i'.inputQueue = N_j.inputQueue$
3. $N_i'.parm = N_j.parm$

*Node* $N_i''$

1. $N_i''.op = N_i.op$
2. $N_i''.inputQueue = N_i'.outputQueue$
3. $N_i''.parm = N_i.parm - N_i'.parm(if\ N_i.op = select)$

   $N_i''.parm = N_i'.parm - N_i.parm(if\ N_i.op = select)$

Consider the *select* nodes of the operator trees of query $Q_6$ and $Q_7$ shown in Figure 5.2. In this case, the *select* node of $Q_7$ is subsumed by the *select* node of $Q_6$. *select* node of $Q_7$ which is the subsumed by the *select* node of $Q_6$ can be decomposed into two *select* nodes. One of these new nodes mirror $Q_6$ and the other is also a *select* node that checks for the additional select condition. Partial sharing is possible because of the overlap of operator trees.

**Definition 10.** *[Overlap of Operator Trees] Two operator trees $OPT(Q_x)$ $OPT(Q_y)$ are said to* overlap *if $OPT(Q_x) \not\equiv OPT(Q_y)$ and there exists a pair of nodes $N_i$ and $N_j$ where $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$ such that $N_i \equiv N_j$.*

**Algorithm 1**: Merge Operator Trees

**INPUT**: $OPT(Q_x)$ and $OPT(Q_y)$
**OUTPUT**: $OPT(Q_{xy})$ representing the merged operator tree

```
1  Initialize N_{Q_xy} = {}
2  Initialize E_{Q_xy} = {}
3  foreach node N_i ∈ N_{Q_x} do
4  |   N_{Q_xy} = N_{Q_xy} ∪ N_i
5  end
6  foreach edge (i, j) ∈ E_{Q_x} do
7  |   E_{Q_xy} = E_{Q_xy} ∪ edge (i, j)
8  end
9  foreach node N_i ∈ N_{Q_y} do
10 |   if ∄N_j ∈ N_{Q_x} such that N_i ≡ N_j then
11 |   |   N_{Q_xy} = N_{Q_xy} ∪ N_i
12 |   end
13 end
14 foreach edge (i, j) ∈ E_{Q_y} do
15 |   if edge (i, j) ∉ E_{Q_xy} then
16 |   |   E_{Q_xy} = E_{Q_xy} ∪ edge (i, j)
17 |   end
18 end
```

When operator trees corresponding to two queries overlap, we can generate the merged operator tree using Algorithm 1. The merged operator tree signifies the processing of the partially shared queries.

Figure 5.3 illustrates the strict sharing of $OPT(Q_4)$ and $OPT(Q_5)$. As shown, we share *select* and *join* operators. The result of the *join* is processed by duplicate preserving project and aggrega-

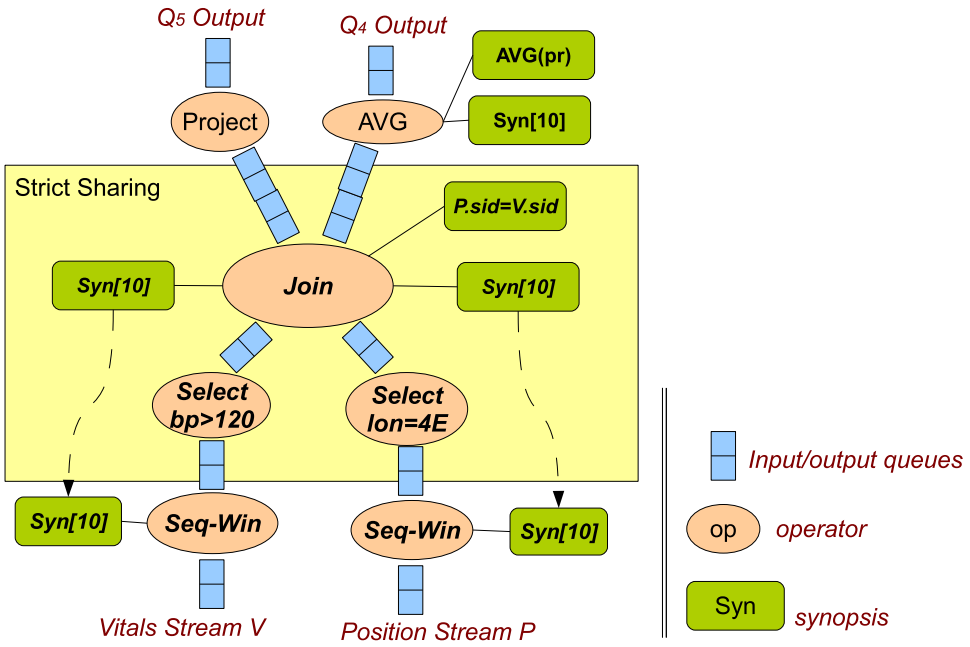


Figure 5.3: Strict Partial Sharing Operator Tree for $\mathtt{Q}_4$ and $\mathtt{Q}_5$

tion operators. On the other hand, seq-window operator is common to all queries using a stream. Figures 5.2 (a) and (b) show the $OPT(Q_6)$ and $OPT(Q_7)$, respectively. Figure 5.2 (c) illustrates the $OPT(Q_{67})$ which shares both the query operations using the loose partial sharing approach. In this case, the query $Q_7$ is subsumed by $Q_6$ according to subsume relation definition. Based on the decomposition of subsumed nodes definition, we split $Q_7$ select condition into two ($bp > 120$ and level = "L") nodes and then share the $bp > 120$ node with $Q_6$.

## 5.4 Scheduling Strategies

A stream processing system handles continuous queries and maintains QoS during bursty input period using scheduling strategies and load shedding techniques [52]. scheduling Researches [26, 11, 13, 45, 10, 22, 72, 27] put efforts to find reasonable execution orders, units, and timings among multiple registered queries. These strategies are critical to a DSMS as they decide CPU allocation schedules to reduce maximal memory requirement and tuple latency, improve throughput, and

avoid starvation.

According to our observations there are three requirements for scheduling strategy design. First, users at dominated levels should not be able to infer about the activities that are taking place at dominating levels. For example, if a low level user can observe that his query is the only one being processed and the throughput is low, he can infer that some high level query is also being processed. Second, illegal information flow should not occur through covert channels because of the sharing of CPU time, memory, and operators across security levels. Third, repeatedly not executing queries at dominating levels may cause a starvation and impact the QoS.

Vanilla STREAM DSMS applies **operator round-robin** in scheduler. Query plan is a linked list of operators. Operators of all query plans from the same user will be collected as a set and they are expected to run consecutively in a round robin manner. After finishing all plans from one user, the scheduler runs queries from another user, and so on. So all registered operators are scheduled to run at least one time per scheduling round. Since DSMS requires fast continuous response for each query, in one round each operator in a plan can process a small fixed number of data tuples (e.g., 100 tuples per round) then switches to other operators. At each heartbeat $\tau$, there will be some tuples buffered in windows for all queries. The scheduler tries to run as many rounds as possible per heartbeat. Because there is no priority among processors at different levels, "first come first serve" strategy is used for plans. The queries from the same user registered first will be executed earliest in each round. When a set of new plans arrives the scheduler under execution, operators will be scheduled in the next execution round.

The regular scheduler runs all operators in every round, and start over the rounds as many as possible. Such mechanism prevents the starvation of late coming queries: each operator is scheduled to process a small amount of inputs in each round. However, this simple round robin

scheduler suffers covert channel threat: the high level users can manipulate the response time of low level queries by adding and reducing high level queries in the scheduler periodically.

To overcome covert channel, **fixed time slot algorithm** is proposed in our previous work [6]. It allocates fixed time slots for each security level and executes corresponding queries within each time slot. If there are no incoming tuples or queries in certain slot, the scheduler remains idle. In this algorithm, idle time slot in low level cannot be used for higher level query execution or vice versa to avoid covert channels. For instance, assume the time duration to be 250 milliseconds. Assume that the scheduler is executing a low level query, and after 50 milliseconds it determines there are no new tuples to process. The scheduler now waits for 200 milliseconds before switching to high level queries. On the other hand, we can start the high level queries after 50 milliseconds. If a new low level tuple arrives after that, it has to wait for 250 milliseconds for the high level processor to complete, before an output is produced by the low level processor. This can reveal unauthorized information to low level users due to inconsistent/varied response times. The fixed time slot method prevents overt and covert channel threats because queries from high level cannot affect the response time of queries in low level. However, the system performance can be significantly affected by the idle time slots.

To improve performance, a simple **revised fixed time slot algorithm** can be done by enforcing new queries can be added to the scheduler at the beginning of each execution round. So the idle time slot in low level can be borrowed by high level queries. Low level user is not able to "sense" high level execution by issuing new queries in his time slot, because all new query registrations will not interrupt running execution round. However, the performance cannot be improved in the case that there are idle slots in high level unavailable to low level queries for security consideration.

## 5.5 Replicated Prototype

We have proposed the replicated prototype in our journal paper [6]. Figure 5.4 shows our replicated architecture for the following experiments. We provide a brief description on the key components.

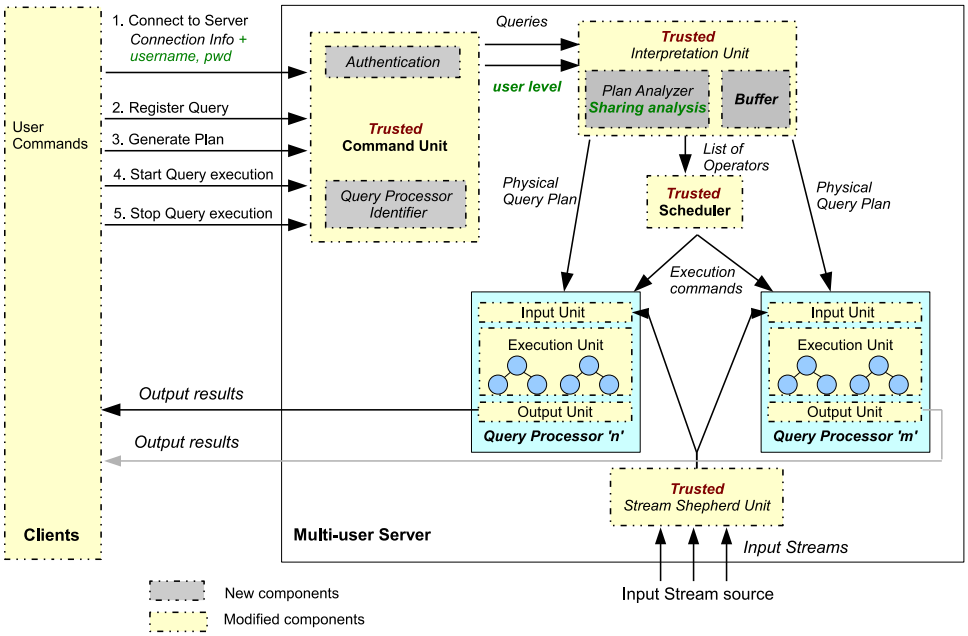


Figure 5.4: Replicated MLS DSMS Architecture

- Trusted command unit is responsible for handling client communication, authentication, and query processor instantiation. It accepts queries from users with different security levels. Each user query needs to be associated with a security level that corresponds to the level at which the query was submitted. The *authentication module* is built in trusted command unit to perform user authentication and security level verification.

  The *query processor identifier* (QPI) module gets users client ID, security level, and query specifications from the authentication module. The QPI maintains the list of currently running query processors. The QPI first checks whether the user queries can be executed in one of the query processors. If Yes, users client ID as well as all the input queries are bound to

that processor. If No, a new query processor is created at that level. We chose this approach to avoid starting query processors if there are no users. The maximum number of query processors is same as the maximum number of security levels supported (4 in our case). The trusted command unit sends the users level and registration information to the trusted interpretation unit. In addition, command unit still controls the query operations like commit and abort with the help of the QPI unit.

- Trusted interpretation unit is responsible for generating query plans and setting up the operators in the scheduler. It receives query information then creates the physical and logical query plans. It sends the physical plans to the proper query processor. The list of operators (also the physical plan) is sent to the trusted scheduler. The three built-in interpretation components (*parser, semantic interpreter,* and *logical plan generator*) and the execution unit of the query processor should be modified accordingly to address MLS-CQL queries.

  The *plan analyzer* checks whether it is possible to share the new coming query with any of the currently executing queries using the buffer. The analysis bases on comparing the semi-product plan of the new and existing plans. The semi-product plan is the semantic analysis result of the query string. User-defined keywords like size of window ,executing operators, output attribute numbers are saved and constructed into the semi-product plan. The *buffer* maintains the query level set information, semi-product, logical and physical plans for all existing queries.

- There are multiple processors and each of them is untrusted, executes at a security level, and has its own input, execution, and output units. The *input unit* can accept input streams from outside sources through trusted stream shepherd unit. It can also accept the output

streams produced from other queries processed by the same query processor; this happens when queries are shared. The *output unit* sends the results back to users continuously.

The *execution unit* is used by the server to execute the physical plans continuously. This unit contains the physical operators and their corresponding algorithms. In order to support MLS, we need to modify the window processing so that it can support filtering conditions based on security levels. The aggregation and join operator algorithms are also revised to compute the least upper bound of the input tuples and use that as the security level of the output tuples. All the operators are untrusted. The execution unit accepts the commands from the trusted scheduler and executes the corresponding operators. There is only one operator running at any point of time, since we have only one scheduler.

- Trusted scheduler will run all queries from different levels using different scheduling strategies. It can run normal round-robin method, or other security-enhanced MLS strategies like fixed time slot.

- Trusted stream shepherd unit handles input streams. In a real-life DSMS, input tuples from different sources can be collected into one multilevel trusted input stream. This unit converts the multilevel trusted streams to single level streams and sends it to the query processor. Besides, load shedding mechanism can be applied here to control the input tuple number under heavy load of data inputs.

# Chapter 6

# Trusted MLS-DSMS

In this chapter, we first present the prototype of trusted MLS-DSMS in order to compare the differences with replicated architecture. Then we explain query rewriting and optimization process specially for trusted DSMS. After that, we present algorithms about query sharing across different levels in trusted system.

## 6.1 Trusted Prototype

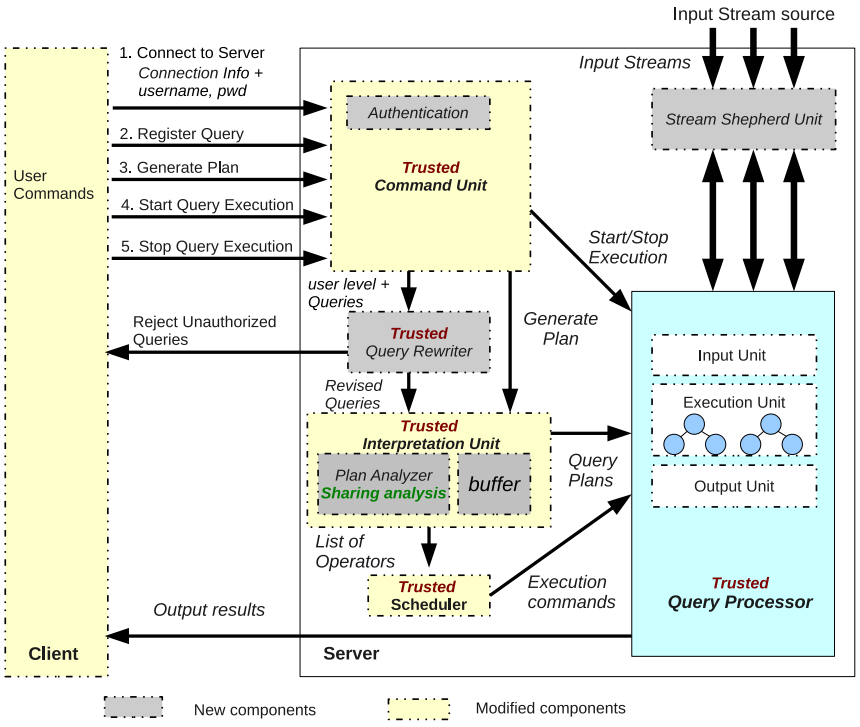


Figure 6.1: Trusted MLS DSMS Architecture

Compared with replicated design, in trusted DSMS there is only one trusted query processor which handles all registered queries in different levels. All queries and input data interact with system components like operators and synopses which are generated by the same processor. So issues like performance improvement and security protection are more complicated than replicated architecture. Now we explain the key components one by one.

***Trusted command unit*** is responsible for handling client communication, authentication, and query processor instantiation. The command unit is trusted and it accepts queries from users with different security levels. Each user query needs to be associated with a security level that corresponds to the level at which the query was submitted. The *authentication module* is built in the trusted command unit to perform user authentication and security level verification. User authenticates by providing user name and password when connecting to the DSMS host. The authentication module then uses this information to retrieve the security level for the particular user. The trusted command unit sends the users (query) level and registration information to the *query rewriter*. In addition, command unit still controls the query operations like commit and abort. After command unit receives first query register request, it initiates a unique trusted query processor. This processor will run all queries from different users.

***Trusted query rewriter*** performs authorization check by comparing requesting data with query level. Since query authorization check is done before plan generation and execution, the system is able to reject unauthorized queries which cause security violations. The query is rewritten and optimized and submitted for further sharing analysis and plan generation in *plan analyzer*. The authorization check, rewriting, and optimization are described in the next section.

***Trusted interpretation unit*** is responsible for generating query plans that involve computation sharing of queries and for setting up the operators in the scheduler. After the *plan analyzer* receives

revised queries from the query sanitizer module, it checks the sharing possibility of the new coming query with any of the currently executing queries' operator(s), such as window, select, project, join, and aggregate operators. The *buffer* in this unit maintains the query level information and execution plans for all existing queries. We have also modified the three built-in interpretation components (*parser, semantic interpreter,* and *logical plan generator*) and the execution unit of the query processor to filter tuples within a window based on security levels.

***Trusted query processor*** run all queries from different users and there is only one in the system. Operators from all queries are prioritized by the trusted scheduler.

***Trusted stream shepherd unit*** handles input streams. In centralized DSMS, input tuples from different sources can be collected into one multilevel trusted input stream. This unit is able to activate load shedding mechanism when the system is under heavy load situation.

## 6.2 Secure Query Rewriting and Optimization

In this section, we discuss how queries submitted by users logged on at a security level are rewritten in a secure form. Recall that queries are submitted by the users logged in at a given security level. The streams that are input to the DSMS, which are referred to as source streams, also have a security level. The individual tuples in the streams are also associated with a security level. Note that, the level of tuple in a stream must be dominated by the level of the stream. The schema of any stream in our model must consist of an attribute, which we refer to as level. The input tuples of the source stream have a level value that is assigned by the source that generates it. The security level of each tuple in the streams generated by DSMS is assigned a value generated by the system. Users can submit queries based on the level, but cannot change its value. In the following examples, we are using four different security levels U<C<S<TS and input streams Vitals and Positions.

Consider the following query $Q_1$.

```
Q1: SELECT AVG(bp) FROM Vitals[Rows 100]
```

The response to this query depends on the level of the user. TS users compute the average blood pressure of 100 tuples at a time, each tuple may belong to any security level. When the same query is issued by a C user, the response consists of the average blood presssure of 100 tuples of soldiers belonging to U or C level.

```
Q1': SELECT AVG(bp) FROM Vitals[Rows 100]
     WHERE level in {U,C};
```

Some may argue that the above query rewriting ($Q_1$') is adequate for providing the response to the C user. However, in this case, some soldiers in the set of 100 who are at level TS or S will not be used for the average computation. Thus, the average will be computed on less than 100 soldiers. This is clearly not desirable. Moreover, this also causes a security breach. It will be possible for users at level C to know how many soldiers are at the S or TS level. Using the available operators in CQL, we cannot express this query. In fact, there are no stream-to-stream operators in CQL.

Recall that we've proposed the stream-to-stream operator that will provide a filter based on the security level and prior to the application of the window operator as demonstrated below.

```
Q1": SELECT AVG(bp) FROM Vitals[Rows 100 level in {U,C}];
```

Often times, queries with non-blocking operators do not have any window clause. Query $Q_2$ illustrates this point.

```
Q2: SELECT bp FROM Vitals;
```

$Q_2$ is equivalent to the following query.

```
Q2': SELECT bp FROM Vitals[Range Unbounded];
```

Consequently, depending on the users who have submitted the query, a filter can be applied based on the security level to the window operations. This filter is automatically added by the system. Thus, if a user at level S issues query $Q_2$, the query rewriting algorithm transforms it into $Q_2$" shown below.

```
Q2": SELECT bp FROM Vitals[Range Unbounded level in {U,C,S}];
```

Users are also allowed to specify the filter condition in the window clause. In such a case, the user specified filter is applied in conjunction with the system specified filter. For example, a user at level TS may specify the following query $Q_3$.

```
Q3: SELECT AVG(bp) FROM Vitals[Rows 100 level in {U,C,TS}];
```

Our query rewriting algorithm will apply the system imposed conjunct based on the security level: {U,C,TS} ∩ {U,C,S,TS} = {U,C,TS}, then transform it into the following query $Q_3'$.

```
Q3': SELECT AVG(bp) FROM Vitals[Rows 100 level in {U,C,TS}];
```

Note that, if a user at level S submitted query $Q_3$, such a query would be denied because the user is requesting information that he is not permitted to view. Algorithm 2 gives our authorization check algorithm. The algorithm takes as its input a query $Q$, its security level $L$, and the security structure for the application that describes the set of levels and their dominance relations. The set of streams in query $Q$ is denoted by the set $Q.stream$, where $Q.stream = \{S_1, S_2, S_3, \ldots, S_n\}$ such that $n \geq 1$. Let $W_i$ be the window associated with stream $S_i$. Let $\sigma_i.level$ be the level clause associated with the selection condition and $W_i.level$ be the set of levels associated with window $W_i$. Note that, for a given stream, $W_i = \{\}$ if there is no window specfied with $S_i$. Similarly, $\sigma_i = \{\}$ if there is no select condition on security levels with respect to stream $S_i$. Lines 1 to 6 compute the set of levels that are dominated by $L$ – we call this set *dominated*. Lines 8 to 10 check

if the set of specified levels in the window clause is a subset of *dominated*. Lines 11 and 12 check is the select condition on the set of levels is a subset of *dominated*. If either of these conditions is violated for any stream, the authorization check fails. Otherwise, it succeeds.

**Algorithm 2**: Authorization Check

**INPUT**: Query($Q$), QueryLevel($L$), Security Structure($\mathbf{L}, \leq$)
**OUTPUT**: Result

```
1  dominated = {}
2  foreach l ∈ L do
3      if l ≤ L then
4          dominated = dominated ∪ {l}
5      end
6  end
7  foreach S_i ∈ Q.Stream do
8      if W_i.level ⊈ dominated then
9          return Authorization Failed
10     end
11     if σ_i.level ⊈ dominated then
12         return Authorization Failed
13     end
14 end
15 return Authorization Passed
```

Consider the following queries submitted by the users at level S. $Q_4$ and $Q_5$ will fail the authorization checks, but $Q_6$ will pass the test.

```
Q4(S): SELECT AVG(bp)
       FROM Vitals[Range 3 minutes level in {TS,C,U}];

Q5(S): SELECT bp FROM Vitals Where level in {C,U,TS};

Q6(S): SELECT AVG(bp)
       FROM Vitals[Range 3 minutes level in {S,C,U}];
```

Once a query successfully passes the authorization tests, the query rewriting algorithm trans-

forms it into a form that ensures that the query can view only the appropriate tuples. Our rewriting algorithm is given in Algorithm 6. Lines 1 – 6 creates the set of levels that are dominated by the query level $L$; this set is referred to as *dominated*. Lines 7 – 10 transform the streams without windows to those with windows where rows are unbounded. In the absence of a filter clause, the $W_i.level$ is assigned the value of *dominated*.

**Algorithm 3**: Secure Query Rewriting

**INPUT**: AuthorizedQuery($Q$), QueryLevel($L$), Security Structure($\mathbf{L}, \leq$)
**OUTPUT**: $Q'$ representing the rewritten query

```
1  dominated = {}
2  foreach l ∈ L do
3      if l ≤ L then
4          dominated = dominated ∪ {l}
5      end
6  end
7  foreach S_i ∈ Q.Stream do
8      if W_i = {} then
9          W_i = Range Unbounded
10     end
11     if W_i.level = {} then
12         W_i.level = dominated
13     end
14 end
```

Consider the following query $Q_7$ submitted by S user. The query rewriting algorithm translates it into $Q'_7$. $Q_7$ and $Q'_7$ are given below.

```
Q7(S): SELECT bp FROM Vitals;

Q7'(S): SELECT bp
        FROM Vitals [Range Unbounded level in {U, C, S}];
```

Once the query is rewritten in a secure form, we need to optimize it for efficient processing.

In addition to the traditional optimizations, we give some new rules involving our new stream-to-stream window operators. We support the different types of windows, each of which gets augmented with a level clause after query rewriting. Recall that, $W_i$ denotes the window of stream $S_i$ and $W_i.level$ gives the set of levels associated with the window. $W_i.type$ denotes the window type where $W_i.type \in \{tuple\_based, time\_based, partition\_by\}$. Each stream $S_i$ may also have a select condition, denoted by $\sigma_i.level$, that filters the result on the basis of security levels. Lines 3 – 5 checks if the intersection of window security levels with those specified in the select condition produces a null set. If so, an error message is returned to the user. Otherwise, depending on the type of window, the window filters and select security level clauses are rewritten. This is done in lines 6 – 13.

**Algorithm 4**: Secure Query Optimization

**INPUT**: SecureQuery($Q$), QueryLevel($L$), Security Structure($\mathbf{L}, \leq$)
**OUTPUT**: $Q'$ representing the optimized query

```
1  foreach S_i ∈ Q.Stream do
2      if σ_i.level ≠ {} then
3          if W_i.level ∩ σ_i.level = {} then
4              return Error: No Output Query
5          end
6          if W_i.type ∈ {time_based, partition_by} then
7              W_i.level = W_i.level ∩ σ_i.level
8              σ_i.level = {}
9          end
10         if W_i.type = tuple_based then
11             if W_i.level ⊆ σ_i.level then
12                 σ_i.level = {}
13             end
14         end
15     end
16 end
```

Consider the following queries submitted by TS users. $Q_8$ returns an error. $Q_9$, $Q_{10}$, and $Q_{11}$ are optimized to $Q'_9$, $Q'_{10}$, and $Q'_{11}$ respectively.

```
Q8(TS): SELECT AVG(bp)
        FROM Vitals[Range 3 minutes level in {TS,C,U}]
        WHERE level in {S};

Q9(TS): SELECT AVG(bp)
        FROM Vitals[Range 3 minutes level in {TS,C,U}]
        WHERE level in {S,C,UC};

Q9'(TS): SELECT AVG(bp)
         FROM Vitals[Range 3 minutes level in {C,U}];

Q10(TS): SELECT AVG(bp) FROM Vitals[Rows 100 level in {C}]
         WHERE level in {C,U};

Q10'(TS): SELECT AVG(bp) FROM Vitals[Rows 100 level in {C}];

Q11(TS): SELECT AVG(bp)
         FROM Vitals[Partition By level in {TS,S,C,U} Rows 100]
         WHERE level in {C};

Q11'(TS): SELECT AVG(bp)
          FROM Vitals[Partition By level in {C} Rows 100];
```

## 6.3 Query Execution and Sharing

Before query execution can proceed, the query plan must be generated. In this work, we represent a query plan in the form of a tree which we refer to as an operator tree defined in Chapter 5.3.1. We assume that the selection condition of the queries are written in a conjunctive normal form. Note that, many operator trees may be associated with a query corresponding to the different plans. However, we show just one such tree for each query.

Table 6.1: Operator Nodes and Specific Parameters

| $N_i.op$ | $N_i.parm$ |
|---|---|
| $select$ | $N_i.parm.cond$ = set of conjuncts of the *select* |
| $proj$ | $N_i.parm.attr$ = set of attributes listed in the *project* |
| $\{avg, count, sum, max, min\}$ | $N_i.parm = \{\}$ |
| $N_i.op = join$ | $N_i.parm.cond$ = set of conjuncts in the *join* |
| $tuple\_win$ | $N_i.parm.row$ = no. of rows<br>and $N_i.parm.level$ = the set of levels of tuples<br>that may be present in the buffer |
| $time\_win$ | $N_i.parm.range$ = time interval for which the tuples<br>and $N_i.parm.level$ = the set of levels of tuples |
| $part\_win$ | $N_i.parm.attr$ = attribute for deriving partitions,<br>$N_i.parm.row$ = no. of rows in the buffer,<br>and $N_i.parm.level$ = the set of security levels of tuples |
| $sel\_win$ | $N_i.parm.level$ = set of levels using for filtering<br>to produce another stream |

In Table 6.1, we describe the different types of nodes of the operator tree and the parameters for each type. The parameters consists of various fields, not all of the fields are applicable to every operator.

Figure 6.2 shows the two operator trees $OPT(Q_x)$ and $OPT(Q_y)$ corresponding to queries $Q_x$ and $Q_y$ as following. Note that the two queries are issued by users in S and TS level respectively.

```
Qx(S):  SELECT COUNT(sid)
        FROM Vitals[Rows 100 level in {S,C,U}]
        WHERE bp > 100 and level in {S};

Qy(TS): SELECT sid
        FROM Vitals[Rows 100 level in {S,C,U}]
        WHERE bp > 100;
```

Queries in a DSMS must be executed efficiently in a resource constrained environment. Thus, if queries can share their computation, we save on the memory and processing costs. Towards this end, we demonstrate how queries can be shared.

We next formalize basic operations that are used for comparing the nodes belonging to different operator trees. Such operations are needed to evaluate whether sharing is possible or not between queries. We begin with the equivalence operator, which is decided by the definition "Equivalence of Nodes". If nodes belonging to different operator trees are equivalent, then only one node needs to be computing for evaluating the queries corresponding to these different operator trees. In Figure 6.2 nodes $OP_1$ and $OP_4$ belonging to $OPT(Q_x)$ and $OPT(Q_y)$ respectively are equivalent.

In some cases, for evaluating node $N_i$ belonging to operator tree $OPT(Q_x)$, we may be able to reuse the results of evaluating node $N_j$ belonging to operator tree $OPT(Q_y)$. This is possible if the nodes are related by the subsumes relationship defined below. Such relationship is possible when the operators match and are non-blocking and the operator parameters are related by a subset relation.

**Definition 11.** *[Subsume Relation of Nodes in Trusted DSMS] Node $N_i \in N_{Q_x}$ is said to be* subsumed *by node $N_j \in N_{Q_y}$, denoted by $N_i \subseteq_T N_j$, where $N_i$, $N_j$ are in the operator trees $OPT(Q_x)$, $OPT(Q_y)$ and are referred to as* subsumed node, subsuming node *respectively, if the following conditions hold:*

1. *Condition 1:*
   - *Case 1 [$N_i.op = project$]:* $N_i.op = N_j.op \wedge N_i.parm \subseteq N_j.parm \wedge N_i.inputQueue = N_j.inputQueue$.
   - *Case 2 [$N_i.op = select$]:* $N_i.op = N_j.op \wedge N_j.parm \subseteq N_i.parm \wedge N_i.inputQueue = N_j.inputQueue$.
   - *Case 3 [$N_i.op = sel\_win$]:* $N_i.op = N_j.op \wedge N_j.parm \subseteq N_i.parm \wedge N_i.inputQueue = N_j.inputQueue$.

- *Case 4 [$N_i.op = tuple\_win$ and $N_j.op = part\_win$]:* $N_i.parm.rows = N_j.parms.rows \wedge$ $N_i.parm.level \subseteq N_j.parm.level \wedge N_i.inputQueue = N_j.inputQueue$

2. *Condition 2: $N_i.op$ is a non-blocking operator.*

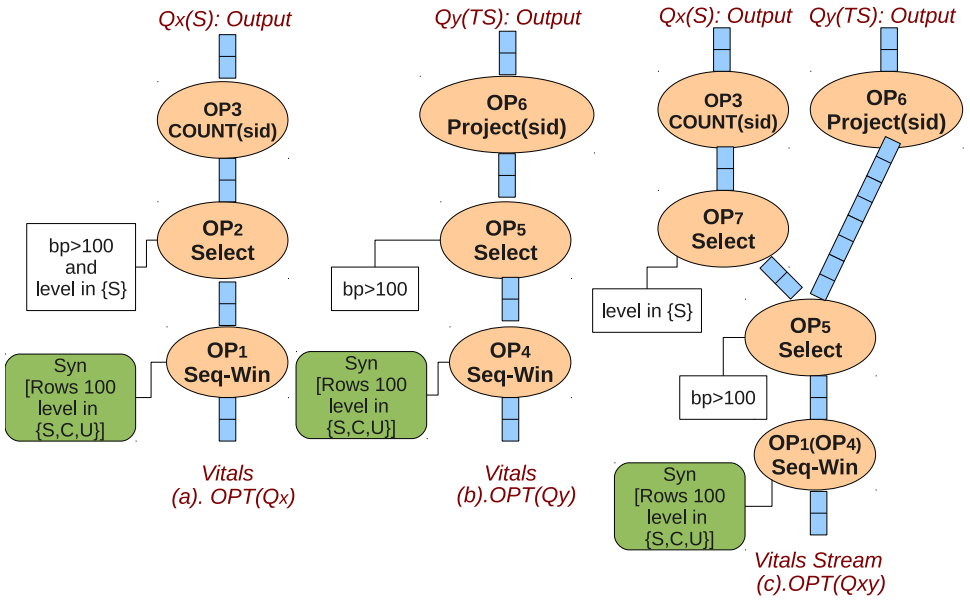


Figure 6.2: Query Sharing

When nodes are related by subsume relation, then it is possible to decompose the subsumed nodes. The decomposition tries to make use of operator evaluation of the subsuming node in order to evaluate the subsumed node. The decomposition is formalized below.

**Definition 12.** *[Decomposition of Subsumed Nodes in Trusted DSMS] Let $N_i \subseteq_T N_j$ where $N_i \in OPT(Q_x)$ and $N_j \in OPT(Q_y)$. Node $N_i$ can be decomposed into two nodes $N_i'$ and $N_i''$ in the following manner.*

*Node $N_i'$*

1. $N'_i.op = N_j.op$
2. $N'_i.inputQueue = N_j.inputQueue$
3. $N'_i.parm = N_j.parm$

*Node* $N''_i$

1. $N''_i.op = N_i.op\ (if\ N_i.op \in \{select, sel\_win, project\})$

   $N''_i.op = select\ (if\ N_i.op = tuple\_win)$
2. $N''_i.inputQueue = N'_i.outputQueue$
3. $N''_i.parm.cond = N_i.parm.cond - N'_i.parm.cond\ (if\ N_i.op \in \{select, sel\_win\})$

   $N''_i.parm.attr = N'_i.parm.attr - N_i.parm.attr\ (if\ N_i.op = project)$

   $N''_i.parm.cond = \{(level \in N_i.parm.level)\}\ (if\ N_i.op = tuple\_win)$

In Figure 6.2 node $OP_2 \subseteq_T OP_5$. $OP_2$ can be decomposed into $OP_5$ and $OP_7$.

When operator trees corresponding to two queries overlap, we can generate the merged operator tree using same merge Algorithm in replicated MLS-DSMS. The merged operator tree signifies the processing of the partially shared queries. Figure 6.2 shows merging of operator trees $OPT(Q_x)$ and $OPT(Q_y)$; the merged operator tree is shown as $OPT(Q_{xy})$.

### 6.3.1 More Sharing Examples

By algorithms of query rewriting and optimization, as well as the definitions described above, the trusted DSMS can now completely or partially share queries across different levels, even between different window operators. Here we show more sharing examples.

**Multilevel Complete Sharing**

We now look at two examples for complete sharing analysis.

```
Qa(TS): SELECT AVG(bp)
        FROM Vitals[Range 3 minutes level in {TS,C,U}]
        WHERE level in {S,C,U};

Qb(TS): SELECT AVG(bp)
        FROM Vitals[Range 3 minutes level in {S,C,U}]
        WHERE level in {C,U};
```

The two queries $Q_a$ and $Q_b$ will be optimized as the following query $Q_{ab}$, so they can be completely shared.

```
Qab(TS): SELECT AVG(bp)
         FROM Vitals[Range 3 minutes level in {C,U}];
```

The following two queries use different kinds of window operators, and they are issued by users in different levels.

```
Qc(TS): SELECT AVG(bp)
        FROM Vitals[Partition By level in {TS,S,C,U} Rows 100]
        WHERE level in {C};

Qd(C):  SELECT AVG(bp) FROM Vitals[Rows 100 level in {C}]
        WHERE level in {C,U};
```

Suppose $Q_c$ is an ongoing query and $Q_d$ just arrives. By queries we can see $Q_c$ computes 4 average results in different levels, and only results in level C will be return. $Q_d$ returns average computation on tuples only in level C. Even $Q_c$ has extra computation, the result is exactly same as $Q_d$. So $Q_c$ and $Q_d$ can be completed shared.

**Multilevel Partial Sharing**

Similar to replicated architecture, trusted MLS-DSMS can also perform strict and loose partial sharing between queries. Strict partial sharing provides rules to share queries using blocking op-

erators like join and aggregation (e.g., average, minimum, maximum, etc). Consider the following two queries $Q_e$ and $Q_f$:

```
Qe(TS): SELECT AVG(bp)
        FROM Vitals[Rows 200 level in {TS,S,C,U}]
        WHERE level in {C,U};

Qf(TS): SELECT AVG(pr)
        FROM Vitals[Rows 200 level in {C,U}]
        WHERE level in {TS,S,C,U};
```

$Q_e$ cannot be partially shared by $Q_f$ because the two tuple-based window operators are neither in equal not subsumed relation. In other words, input tuple expiration for $Q_e$ can be triggered when a new tuple in level TS and S comes, while $Q_f$ does not. So the average computations between two queries use different set of buffered 200 tuples.

Let us consider another pair of queries $Q_g$ and $Q_h$:

```
Qg(S):  SELECT V.sid, bp, pr
        FROM Vitals[Range 2 Minutes level in {S,U}] V,
        Position[Range 2 Minutes level in {S,U}] P
        WHERE V.sid = P.sid AND level in {C,U};

Qh(S):  SELECT V.sid, lat, lon
        FROM Vitals[Range 2 Minutes level in {C,U}] V,
        Position[Range 2 Minutes in {C,U}] P
        WHERE V.sid = P.sid AND level in {S,U};
```

After query rewriting and optimization, $Q_g$ and $Q_h$ are transformed as following queries $Q'_g$ and $Q'_h$. The join operation can be partially shared between them because there are no subsumed relation between window operators by Definition 11.

```
Qg'(S): SELECT V.sid, bp, pr
        FROM Vitals[Range 2 Minutes level in {U}] V,
```

```
        Position[Range 2 Minutes level in {U}] P
        WHERE V.sid = P.sid;

Qh'(S): SELECT V.sid, lat, lon
        FROM Vitals[Range 2 Minutes level in {U}] V,
        Position[Range 2 Minutes in {U}] P
        WHERE V.sid = P.sid;
```

Non-blocking operators such as selection and projection can be shared in loose partial sharing. We try to reuse partial processing results from another query with higher level section in where clause specification. Here is an example for loose partial sharing.

```
Qi(S):  SELECT sid, bp FROM Vitals[Rows 100 level in {C,U}]
        WHERE bp > 120 AND level in {C,S};

Qj(TS): SELECT sid, pr FROM Vitals[Rows 100 level in {S,C,U}]
        Where bp > 120 AND bp < 180 AND level in {C};
```

After query rewriting, the two queries are changed to $Q'_i$ and $Q'_j$.

```
Qi'(S):  SELECT sid, bp FROM Vitals[Rows 100 level in {C}]
         WHERE bp > 120;

Qj'(TS): SELECT sid, pr FROM Vitals[Rows 100 level in {C}]
         WHERE bp > 120 AND bp < 180;
```

$Q'_i$ can be partially shared by $Q'_j$ because select operators $\mathrm{S}_{Q'_j} \subseteq_T \mathrm{S}_{Q'_i}$.

## 6.4 Scheduling Methods

To further improve performance without security violation, we are proposing **round-robin with dynamic threshold control algorithm**. The scheduler first reorders all queries from lowest to highest level, then execute the operators of particular level with a fix small amount of inputs in each round like round-robin.

A threshold control is to detect and change the execution way of some operators in certain level. If those operators consume most execution time in scheduler which causes starvation problem or potential covert channels (e.g., takes 80% of execution time per round), all operators in that level will be given two set of **penalties**: *percentage penalty*, the percentage of the operators in certain level can be scheduled in each round; *round penalty*, the number of rounds of the penalty will last. We use two random values $\alpha$ (e.g., from 0.3 to 0.8) and $\beta$ (e.g., from 1 to 10) as the two penalty parameters respectively. Suppose there are 10 operators from $O_1$ to $O_{10}$ in the scheduler. $O_1$ and $Q_2$ are from high level queries and the others are from low level queries issued by a malicious user. Suppose each operator takes equal time for executing the same amount inputs, and the scheduler will execute those operators as following rounds:

- In round 1, scheduler runs all 10 operators and notices that operators in low level consumes 80% execution time. So all low level operators are put into penalty set and only part of them (e.g, half $\alpha$ = 0.5) can be executed in each round (only 4 of 8 operators from $O_3$ to $O_{10}$ will scheduled in next round). Besides, the penalty will last 10 rounds ($\beta$ = 10).

- In round 2, scheduler runs operators from $O_1$ to $O_6$. While $O_7$ to $O_{10}$ are hold without execution.

- In round 3, scheduler runs operators $O_1$, $O_2$ and $O_7$ to $O_{10}$. Suppose the malicious low level user issues new queries with 4 operators $O_{11}$ to $O_{14}$. The four queries are added to the low level penalty queue, and parameters $\alpha$ and $\beta$ are changed (e.g., $\alpha$ = 0.25, $\beta$ = 6).

- In round 4, since $\alpha$ = 0.25, there are 12 * 0.25 = 3 operators from penalty set will be scheduled. So the scheduler runs $O_1$, $O_2$ and $O_{11}$ to $O_{13}$. If there is more low level queries come, the two parameters will be changed with random values.

By this algorithm two parameters $\alpha$ and $\beta$ are changed when there is a query coming or cancellation. Note that once the penalty set is established, the penalty will continuous for certain rounds even though the operator number drops below threshold $\alpha$ by cancelling some queries in this level. The consistent penalty and dynamic threshold prevent covert channels where high level queries manipulate the response time of low level query by adding/cancelling numbers of high level queries in certain time pattern. Besides, the performances can be improved since time slots by levels are cancelled and there is no idle slot for specific level during execution.

On the other hand, this method is not perfect because penalties are applied to all queries in certain level if one of them violates threshold control. Such approach prevents covert channel threats but affect the performances of other legal queries. In future work we plan to find a better detection algorithm can pinpoint the malicious user and suspend only those queries in the scheduler. In current trusted MLS-DSMS implementation, we use round-robin scheduling method in order to provide better performance.

# Chapter 7

# Distributed MLS-DSMS

System availability is one of three fundamental requirements in database security. In this chapter, we propose a simple distributed MLS-DSMS. This work has been presented in our newest ASDN'13 paper [76].

In our distributed model, we assume there is a set of servers each with its unique id. Each server has a preassigned security level which is never changed. The set of servers having the same security level forms a group. Each server maintains a list of authorized users for that level. Consequently, a client can submit his query to any server at the particular level. The server to which a client submits his query is referred to as the *master*. The other servers in the same level act as *slaves*. The master coordinates the execution of the query. It receives the results from the slaves and forwards them to the user. The members in a group communicate with each other with respect to their load and status. Consequently, the master is able to achieve load balancing for the given queries.

## 7.1 Prototype Implementation

In the following, we describe our prototype implementation.

### 7.1.1 Server Communications

Each startup DSMS server has a unique server ID ($sid$) as IP address and the port number, as well as an pre-assigned security level $L_s$. Online servers with same security level form a *group*

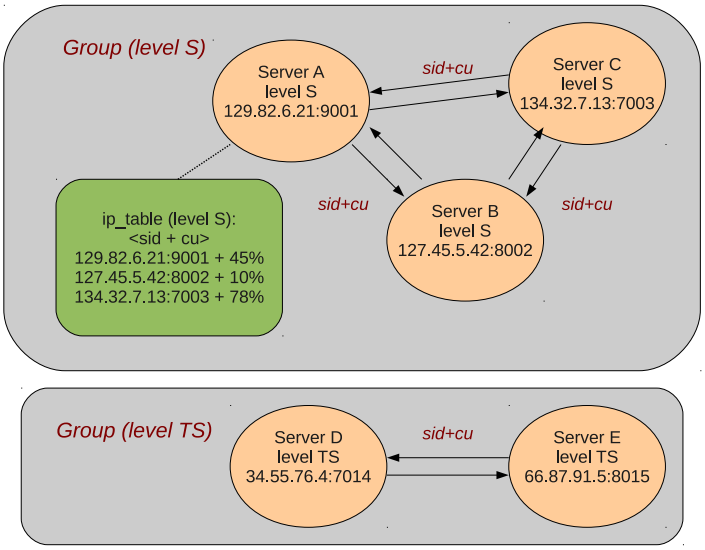


Figure 7.1: Group Construction

for communication. Server information like $sid$ and current cpu usage ($cu$) are sent to a particular multicast *group* continuously. The servers are listening to others in the same group for getting their peers' information.

Communications between servers are allowed only if they are in the same group. An *ip_table* recording the $sid$ and $cu$ of available members in the group are kept in each server. Since the availability and CPU usage of servers are changed in real-time, the *ip_table* is also updated accordingly. Figure 7.1 gives an example of how the two groups are constructed. Servers A, B and C in the same level forms a group in level S. They are exchanging the server info to each other. Each server maintains an *ip_table* contains all server info in the group. Servers D and E forms another group in TS level. The two groups are disjoint.

### 7.1.2 Distributed Processing

Once a group is constructed, any server inside can handle the request from clients in the same security level. Now we explain distributed processing step by step.

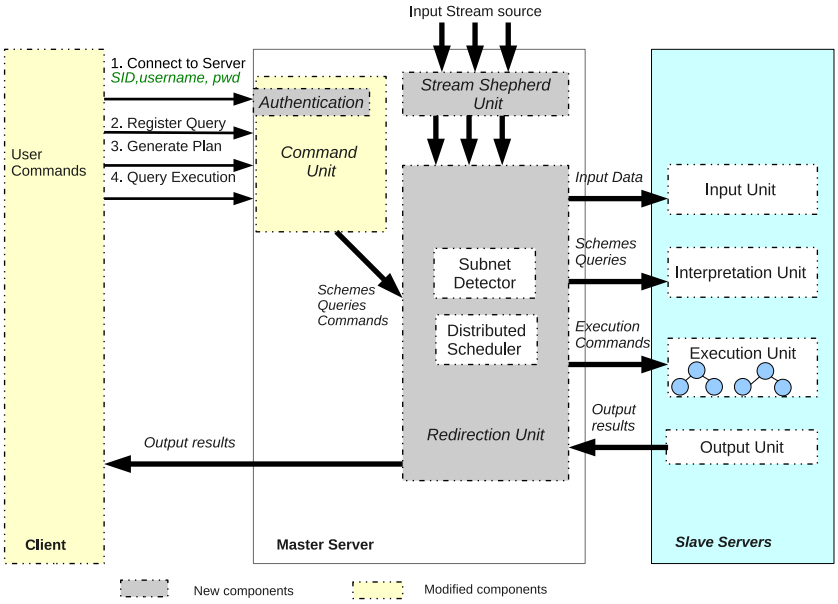


Figure 7.2: Distributed DSMS Architecture

**Client Connection:**

Our prototype is showed in Figure 7.2. Each server maintains a profile of authorized users whose login level is identical to the server level. Authentication is performed when a user runs a client software to connect to a server. Only the user whose username and password are present in the profile are authorized for further operations. Recall that the server to which a user successfully connects is called the master server. Other servers in the same group as the master are referred to as slaves. Master acts as a dispatcher and distributes the query load for execution to slave servers.

**Query Registration:**

Once a user has been authenticated, the master uses all the available members in the group to act as slaves and provide processing power for queries submitted by user. The client begins to register input stream schemes in master and all slave servers. Besides, a query registration message from client is redirected to the *interpretation unit* in slaves which translates the interpreted query to

a logical query plan (a link of operators). The naive physical plan is also generated. It then redirects the user entering info to all available members in the group and sends registration and query generation commands to all slaves. Note that once the user begins registration, master would not allow new server to participate in the processing of queries submitted by this user.

The incoming streams are connected to the query redirection unit by the *stream shepherd unit* in master. The stream shepherd unit is trusted and it will filter out tuples from the strictly dominating level before sending it to the redirection unit. For example, the TS level input tuple will not enter the redirection unit of master in level S. Output connection between master and client is established after the queries are registered.

**Plan Generation:**

Once the master receives command from the client indicating that all queries have been registered, it requests all slaves for optimizing the naive physical query plans created in the previous step. Also, graphs of physical plans are generated for user view. The generated physical plans are instantiated in the execution unit of all slave servers.

**Query execution:**

Once the *start query execution command* is issued, master starts *distributed scheduler* to dispatch loads in different slave servers. In our DSMS system, all data that arrives within one second are taken as an *input chunk* which is handled by one slave. For simple queries without window specification, the master delivers the whole chunk to a slave, waits and receives the execution results which it sends back to the user.

Algorithm 5 is running continuously in master during query execution. Group detector and distributed scheduler in our prototype coordinate together perform load distribution. There are two

major functions in the distribution algorithm:

- The master reads CPU usage $cu$ from every slave server. If the slave is not in heavy load (in our case $cu$ is smaller than 90%), the master can deliver a chunk of input data to the slave. Otherwise, the "busy" slave is not considered to participate in this computation round.

- In every 1 minute, master server will update the existence and CPU usage information of slaves in its own *ip_table*. Offline servers will be deleted from this round and new $cu$ values are used for next load distribution scheduling.

Master acts as a dispatcher and distribute the query processing workload to slaves – this is, however, transparent to the user.

## 7.2 Input Chuck Construction

CQL window operators are frequently used in streaming processing system. They are generated as the bottom most operators in a query plan, and used to buffer input tuples within a fixed size or a time period for further processing. For computations of blocking operators such as join and aggregation, the accuracy of processing results relies on previous and new coming inputs stored in the buffer. In distributed DSMS, however, the inputs are divided as *isolated* chunks then sent to different slave machines. To preserve the data continuity in buffer between different chucks, our distributed DSMS combines extra data with the current input data to current scheduled slave.

**Definition 13.** *[Input Chuck Construction for Window Operators] For a CQL query using window operator with input stream $S$, an input chuck $\mathcal{I}_S = I_S^w + I_S^c$, where $I_S^w = \Sigma S_i(N_i)$, $S_i(N_i)$ represents the last $N$ inputs from stream $S$ in timestamp $i$; $I_S^c$ is the new data arrived in current timestamp.*

With the construction formula, we explain how input chucks are constructed by three example queries. There are three queries $Q_a$, $q_B$, $Q_c$ submitted in distributed DSMS systems. From timestamp 1 to 4, the numbers of input data in Vitals $V$ stream are 200, 50, 40, 70. And the numbers of input data in Positions $P$ stream are 50, 30, 10, 80.

```
Qa: SELECT AVG(bp) FROM Vitals[ROWS 100];
```

```
Qb: SELECT (bp) FROM Vitals[RANGE 2 Seconds];
```

```
Qc: SELECT * FROM Vitals[ROWS 80], Positions[ROWS 80];
```

From Table 7.1, tuple-based window operator in master system should keep the part of previous inputs received in earlier timestamps. For time-based window, the master buffers all inputs within time period in window specification.

To illustrate the computation details in different slaves, let us see another query $Q_d$ runs in 1 master and 2 slave machines $S_1$ and $S_2$. In timestamp 1 and 3, the new coming inputs are redirected to $S_1$ while in timestamp 2 the inputs are sent to $S_2$.

```
Qd: Select AVG(bp) from Vitals[Rows 3];
```

From Table 7.2, in all timestamps (ts) except ts = 1, we can see there is an extra legacy chuck construction step before computation. The reason is obvious because the inputs from previous timestamps are meaningful in current stage. With this chuck construction for every running slave server, the outputs can be correctly sent back to user.

**Algorithm 5**: DSMS Load Distribution

**INPUT**: master $m$, slave set $S$, current CPU usage $cu$, server info $sid$, refresh timer $t$, IP multicasting message in group

```
1  Read slave set S from m's ip_table;
2  Start refresh timer t;
3  while StopExecution == false do
4      foreach slave server s ∈ S do
5          if s → cu ≤ 90% then
6              m distribute an input chunk to address s → sid
7          end
8          else
9              Skip load distribution to address s → sid
10         end
11     end
12     if t ≥ 1 minute then
13         foreach slave server s ∈ S do
14             Read IP multicasting messages from the group
15             if s → sid is not found then
16                 Delete s from ip_table in m
17             end
18             else
19                 Update s → cu using the updated info from the message
20             end
21         end
22         Restart timer t
23     end
24 end
```

Table 7.1: Input Chucks Construction Examples

| | $Q_a$ | $Q_b$ | $Q_c$ |
|---|---|---|---|
| ts=1, $V$=200, $P$=50 | $\mathcal{I}_V$=200 $I_S^w$=0, $I_S^c$=200 | $\mathcal{I}_V$=200 $I_S^w$=0, $I_S^c$=200 | $\mathcal{I}_V$=200, $\mathcal{I}_P$=50 $I_S^w$=0, $I_S^c$=200 $I_P^w$=0, $I_P^c$=50 |
| ts=2, $V$=50, $P$=30 | $\mathcal{I}_V$=150 $I_S^w$=$V_1(100)$, $I_S^c$=50 | $\mathcal{I}_V$=250 $I_S^w$=$V_1(200)$, $I_S^c$=50 | $\mathcal{I}_V$=130, $\mathcal{I}_P$=80 $I_S^w$=$V_1(80)$, $I_S^c$=50 $I_P^w$=$P_1(50)$, $I_P^c$=30 |
| ts=3, $V$=40, $P$=10 | $\mathcal{I}_V$=140 $I_S^w$=$V_1(50)$+$V_2(50)$, $I_S^c$=40 | $\mathcal{I}_V$=90 $I_S^w$=$V_2(50)$, $I_S^c$=40 | $\mathcal{I}_V$=120, $\mathcal{I}_P$=90 $I_S^w$=$V_1(30)$+$V_2(50)$, $I_S^c$=40 $I_P^w$=$P_1(50)$+$P_2(30)$, $I_P^c$=10 |
| ts=4, $V$=70, $P$=80 | $\mathcal{I}_V$=170 $I_S^w$=$V_1(10)$+$V_2(50)$+$V_3(40)$, $I_S^c$=70 | $\mathcal{I}_V$=110 $I_S^w$=$V_3(40)$, $I_S^c$=70 | $\mathcal{I}_V$=150, $\mathcal{I}_P$=160 $I_S^w$=$V_2(40)$+$V_3(40)$, $I_S^c$=70 $I_P^w$=$P_1(50)$+$P_2(30)$+$P_3(10)$, $I_P^c$=80 |

Table 7.2: Computation in Different Slaves

| Input(ts, sid, bp,slave) | Buffer(ts,sid,bp) | AVG(sign,result) |
|---|---|---|
| (1,AAA,125,$S_1$) | (1,AAA,125) | (+,125) |
| (1,BBB,100, $S_1$) | (1,AAA,125),(1,BBB,100) | (-,125),(+,112.5) |
| (1,CCC,150, $S_1$) | (1,AAA,125),(1,BBB,100),(1,CCC,150) | (-,112.5), (+,125) |
| Legacy Buffer Construction (2,DDD,110, $S_2$) | (1,AAA,125),(1,BBB,100),(1,CCC,150) (1,BBB,100),(1,CCC,150),(2,DDD,110) | (-,125), (+,120) |
| Legacy Buffer Construction (3,EEE,160,$S_1$) | (1,BBB,100),(1,CCC,150),(2,DDD,110) (1,CCC,150),(2,DDD,110),(3,EEE,160) | (-,120), (+,140) |

# Chapter 8

# Stream Audit Cloud Application

In this chapter, we extend MLS-DSMS to a secure stream audit applications. We are proposing an information flow control model adapted from the Chinese Wall policy [58] that can be used to provide secure processing of streaming data generated from multiple organizations. The work of CW-DSMS development has been presented in our newest SACMAT'13 paper [75].

A cloud contains a set of companies that offer services. In order to keep the cloud operational, it is important to detect security and performance problems in a timely manner. Thus, auditing live events streaming from the cloud is very essential. Services offered in a cloud can be competing or complementing. To detect attacks and performance issues, the cloud has to be audited as a whole, though the audit events may be generated by competing or complementing companies. Chinese Wall policy aims to protect disclosure of company sensitive information to potentially competing organizations, but does not deal with complementing organizations. In a cloud, companies are organized into various domains based on the types of services they provide. Each of these domains forms a *conflict of interest* (COI) class. Companies in the same COI class are in direct competition. We must aim to prevent leakage of a company's sensitive information to other organizations belonging to the same COI class. Companies that offer complementing services can be assigned a complementing interest (CI) class. Companies in the same COI class cannot be in the same CI. Companies belonging to the same CI have no such direct competition and do not require trusted entities to manage their information.

Streaming audit data generated by various organizations must be analyzed in real-time to detect the presence of various types of attacks. A company may want to audit its own data to detect malicious insider threats. Sometimes it may be needed to detect a denial-of-service attack for a particular type of service offered by companies in a COI class. On the other hand, detecting the delay between the service request and response may involve analyzing audit streams in a service chain invocation that has multiple companies belonging to some CI class. For each such case, it should be possible to detect the attack without causing a company's sensitive information from being leaked to its competitors.

To address secure stream auditing, we start with identifying the access requirements and the information flow constraints for processing streaming audit data in a cloud computing environment. We first adapt the Chinese Wall policy formulated by Sandhu [58] to formalize the information flow constraints in clouds. Then we demonstrate how cloud computing queries can be formulated and provide an architecture for executing such queries, using some ideas from replicated MLS-DSMS. By applying the sharing strategies from MLS-DSMS system, we also implement a prototype to demonstrate the feasibility of our approach and show how the performance is impacted by the information flow constraints.

## 8.1 Information Flow Model

In the following, we present an information flow model for cloud applications to protect against improper leakage and disclosure. We provide an information flow model that is adapted from the lattice structure for Chinese Wall proposed by Sandhu [58].

We have a set of companies that provide services in the clouds. The companies are partitioned into conflict of interest classes based on the type of services they provide. Companies providing

the same type of service are in direct competition with each other. Consequently, it is important to protect against disclosure of sensitive information to competing organizations. We begin by defining how the conflict of interest classes are represented.

**Definition 14. [Conflict of Interest Class Representation:]** *The set of companies providing service to the cloud are partitioned into a set of $n$ conflict of interest classes, which we denote by $COI_1$, $COI_2$, ..., and $COI_n$. Each conflict of interest class $COI_i$ consists of $m_i$ companies, where $m_i \geq 1$, that is $COI_i = \{1, 2, 3, \ldots, m_i\}$.*

A set of companies, who are not in competition with each other, provide complementing services in the cloud. A single company can provide some service, and sometimes multiple companies may together offer a set of services. In the following, we define the notion of complementing interest (CI) class and show how it is represented.

**Definition 15. [Complementing Interest Class Representation:]** *The set of companies providing complementing services is represented as an n-element vector $[i_1, i_2, \ldots, i_n]$, where $i_k \in COI_k \cup \{\perp\}$. $i_k = \perp$ signifies that it does not contain services from any company in $COI_k$. $i_k \in COI_k$ indicates that it contains services from the corresponding company in $COI_k$. Our representation forbids multiple companies that are part of the same COI class from being assigned to the same complementing interest class.*

We next define the security structure of our model. Each data stream, as well as the individual tuples constituting it, is associated with a security level that captures its sensitivity. Security level associated with a data stream dictates which entities can access or modify it. Input data stream generated by an individual organization offering some service has a security level that captures the organizational information. Input streams may be processed by the DSMS to generate *derived*

*streams*. Derived data streams may contain information about multiple companies, some of which are in the same COI class and others may belong to different COI classes. Before describing how to assign security levels to derived data streams, we show how security levels are represented.

**Definition 16. [Security Level Representation:]** *A security level is represented as an $n$-element vector $[i_1, i_2, \ldots, i_n]$, where $i_j \in COI_j \cup \{\perp\} \cup \{T\}$. $i_j = \perp$ signifies that it does not contain information from any company in $COI_j$; $i_j = T$ signifies that the data stream contains information from two or more companies belonging to $COI_j$; $i_j \in COI_j$ denotes that it contains information from the corresponding company in $COI_j$.*

Consider the case where we have 3 COI classes, namely, $COI_1$, $COI_2$, and $COI_3$. $COI_1$, $COI_2$, and $COI_3$ have 5, 3, and 2 companies, respectively. The audit stream generated by Company 5 in $COI_1$ has a security level of $[5, \perp, \perp]$. Similarly, the audit stream generated by Company 2 in $COI_3$ has a security level $[\perp, \perp, 2]$. When audit streams generated from multiple companies are combined, the information contained in this derived stream has a higher security level. For example, audit stream having level $[5, \perp, 2]$ contains information about Company 5 in $COI_1$ and Company 2 in $COI_3$. It is also possible for audit streams to have information from multiple companies belonging to the same $COI$ class. For example, a security level of $[5, \perp, T]$ indicates that the data stream has information from Company 5 in $COI_1$, does not contain information from $COI_2$, and information about multiple companies in $COI_3$. We also have a level $[\perp, \perp, \perp]$ which we call *public* and that has no company specific information. The level $[T, T, T]$ correspond to level *trusted* and it contains information pertaining to multiple companies in each COI class and can be only accessed by trusted entities. We next define dominance relation between security levels.

**Definition 17. [Dominance Relation:]** *Let* **L** *be the set of security levels, $L_1$ and $L_2$ be two*

*security levels, where* $L_1, L_2 \in \mathbf{L}$. *We say security level* $L_1$ *is dominated by* $L_2$*, denoted by* $L_1 \preceq L_2$*, when the following conditions hold: (*$\forall i_k = 1, 2, \ldots, n)(L_1[i_k] = L_2[i_k] \vee L_1[i_k] = \perp \vee L_2[i_k] = T$*). For any two levels,* $L_p$, $L_q \in \mathbf{L}$*, if neither* $L_p \preceq L_q$*, nor* $L_q \preceq L_p$*, we say that* $L_p$ *and* $L_q$ *are incomparable.*

The dominance relation is reflexive, antisymmetric, and transitive. The level *public*, denoted by $[\perp, \perp, \perp]$, is dominated by all the other levels. Similarly, the level *trusted*, denoted by $[T, T, T]$, dominates all the other levels. Note that the dominance relation defines a lattice structure, where level *public* appears at the bottom and the level *trusted* appears at the top. Incomparable levels are not connected in this lattice structure. In our earlier example, level $[5, \perp, \perp]$ is dominated by $[5, \perp, 2]$ and $[5, \perp, T]$. $[5, \perp, 2]$ is dominated by $[5, \perp, T]$. That is, $[5, \perp, \perp] \preceq [5, \perp, 2]$ and $[5, \perp, 2] \preceq [5, \perp, T]$. $[5, \perp, \perp]$ and $[\perp, \perp, 2]$ are incomparable.

Each data stream is assigned a security level. Each of the tuples constituting the data stream also has a security level assigned to it. The security level of the individual tuples in a data stream are dominated by the level of the data stream. When a DSMS operation is executed on multiple input tuples, each having its own security level, an output tuple is produced. The security level of the output tuple is the least upper bound (LUB) of the security levels of the input tuples.

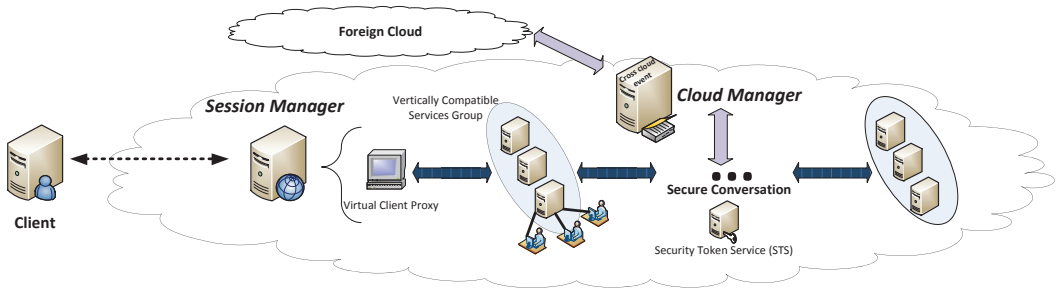


Figure 8.1: Multi-Tier Architecture of a Cloud

In audit application, various types of queries are executed to detect security and performance problems. Each continuous query $Q_i$, submitted by a process, inherits the security level of the process. Similar to MLS-DSMS, we require a query $Q_i$ to obey the simple security property and the restricted $\star$-property of the Bell-Lapadula model [16].

1. Query $Q_i$ with $L(Q_i) = C$ can read a data stream $x$ only if $L(x) \preceq C$.

2. Query $Q_i$ with $L(OP_i) = C$ can write a data stream $x$ only if $L(x) = C$.

Note that, for our example, a process executing at level $[5, \perp, T]$ can execute streams belonging to Company 5 in $COI_1$ and all companies in $COI_3$ and also streams derived from them. Thus, the process is trusted w.r.t. $COI_3$, but not w.r.t. the other COI classes. Our information flow model thus provides a finer granularity of trust than provided by the earlier models. Our goal is to allow information flow only from the dominated levels to the dominating ones. All other information flow, either overtly or covertly, should be disallowed by our architecture.

## 8.2 Continuous Query Processing Architecture

In this section, we present our example application that motivates the need for secure stream processing in cloud computing environments. We have a service that aims to prevent and detect attacks in real-time in the cloud. Such a service provides warning about various types of attacks, often involving multiple organizations.

Figure 8.1 shows a multi tier architecture of the cloud adopted from [74]. Various types of auditing may take place in the cloud. The first level is the *company auditing tier*, not explicitly shown in Figure 8.1, is represented by the users connected to some service. In this tier, the activities pertaining to an organization are analyzed in isolation. The next level is the *service auditing*

*tier*, identified by shaded ellipses that contain sets of resources and services. Each shaded ellipse depicts vertically compatible services or resources; this implies the services or resources that can be functionally substituted for each other, possibly on demand. The *cloud auditing tier* is shown with connecting dark arrows, which depicts the internal communication within the cloud due to a service invocation chain.

Various types of audit streams must be captured to detect the different types of attacks that may take place in a cloud. The company auditing tier logs the activities of the various users in the organization. If the behavior of an authorized user does not follow his usual pattern, we can perform analysis to determine if the user's authentication information has been compromised. This tier is responsible for analyzing the audit streams of individual companies in isolation. Typically, at this layer, the audit streams generated by a single company are analyzed.

The service auditing tier logs information pertaining to the various companies who provide similar services. Session Manager at this tier can detect whether there is a denial-of-service attack targeted at a specific type of service. Session Manager analyzes audit streams generated from multiple competing organizations, so we need to protect against information leakage and corruption. In short, the Session Manager needs to analyze data from one or more companies belonging to the same COI class.

The cloud auditing tier collects audit information pertaining to a service invocation chain and is able to detect the presence of man-in-the-middle attack. Cloud Provider is responsible for analyzing audit streams from multiple organizations associated with service invocation chains, but the organizations may not have conflict of interest. Thus, at this tier, the audit streams from the companies belonging to one or more CI classes are analyzed.

In order to detect and warn against these attacks, continuous queries must be executed on the

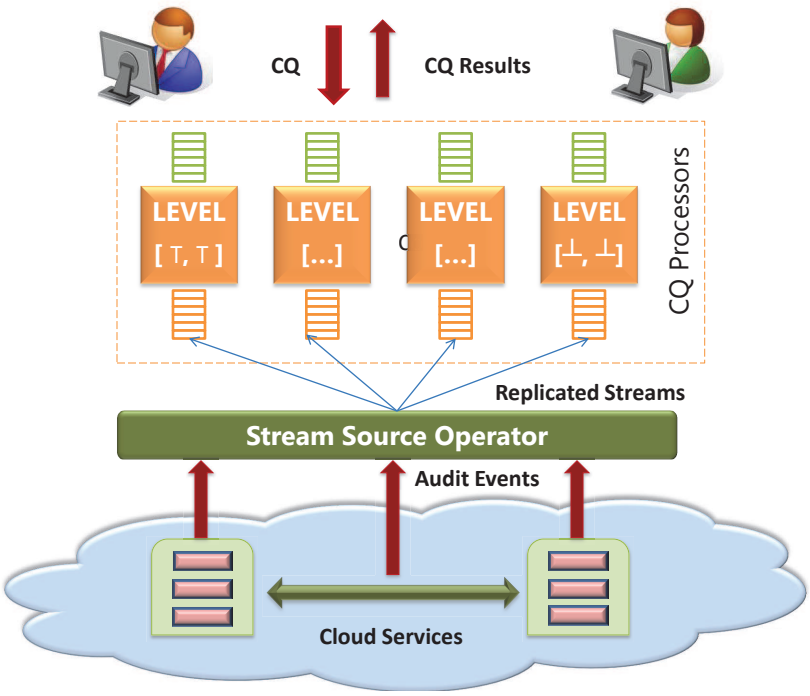


Figure 8.2: CQ Processing Architecture

streaming data belonging to various organizations. Queries must be processed such that there are no overt or covert leakage of information across competing organizations.

We propose the architecture shown in Figure 8.2 that provides a way to capture events from the cloud, monitor them, and trigger alerts. The architecture is based on cloud computing [74], data stream processing [26, 12, 8, 30], event processing [25, 3], Chinese wall security [58], replicated and trusted multilevel database management [2] and replicated MLS-DSMS in previous chapter.

As shown in Figure 8.2 there are several services offered in the cloud. Data generated by these servers are propogated to the DSMS. For this paper, we consider a centralized DSMS architecture. Compatible services are grouped and they interact based on client needs. Each of the service and other servers contain an event detector to monitor and detect occurrence of interests. The detectors sanitize and propagate the events to the data stream management system, which arrive at the stream

source operator. This operator checks for the level of the incoming audit events and propagates them to the appropriate query processor's input queue. The query processor architecture is based on the replicated model, where there is a one-to-one correspondence between query processors and security levels. A query specified by a user at a particular level is executed by the query processor running at that level. Also the query processor can only process data that are dominated by the query processor level. This replicated approach allows the use of untrusted query processors. After processing the query results are disseminated to authorized users via the output queues of queries. In addition to the query processors and stream source operator the data stream management system contains various other components (trusted and untrusted) as discussed in the implementation and experimental evaluation section.

Other alternative architectures include trusted and hybrid. In the trusted architecture there will be only one query processor. This query processor is trusted and all continuous queries are executed in this processor. This architecture will have less administrative overhead and also useful during sharing of continuous queries. The disadvantages include creating trusted code is hard, and threat of covert and overt channels. In the hybrid architecture, we can interleave the queries based on the CIs and also have separate processor for handling trusted and public tuples. In this work we consider the replicated architecture as the first step to run it in stream audit application.

## 8.3 Query Processing in Cloud DSMS

In this section, first we discuss the different types of queries that can be executed on cloud audit data at the different tiers.

### 8.3.1 Cloud CQL Queries

Consider a simple application that tries to detect example denial-of-service attacks in the cloud. We have two conflict of interest classes denoted by $COI_1$ and $COI_2$. The constituent companies in each COI class is given by, $COI_1 = \{1, 2\}$ and $COI_2 = \{A, B, C\}$. Examples of security levels in our configuration are $[\perp, \perp]$ (public knowledge), $[T, T]$ (completely trusted), $[1, \perp]$ (data from 1), $[\perp, T]$ (trusted w.r.t. $COI_2$), $[1, B]$ (data from 1 and $B$), $[1, T]$ (data from 1 in $COI_1$ and trusted w.r.t. $COI_2$). Continuous queries are executed at various tiers to detect performance delays and possibly denial-of-service (DoS) attacks. In any given tier, different types of DoS attacks may occur – some involving the data belonging to single organizations, others involving data belonging to multiple organizations. Thus, a tier can have query processors at different levels, each of which executes queries on data that it is authorized to view and modify.

We consider a single data stream, called `MessageLog`, that contains the audit stream data associated with message events, such as `send` and `receive`. `MessageLog` is obtained from `SystemLog` by filtering the events related to sending and receiving the messages. Note that, `MessageLog` in reality may contain many other fields, but we only deal with those that are pertinent to this example. The various attributes in `MessageLog` are `serviceId`, `msgType`, `sender`, `receiver`, `timestamp`, `outcome`. `serviceId` is a unique identifier associated with each service; `msgType` gives the type of message which is either `send` or `receive`; `sender` (`receiver`) gives the id of the organization sending (receiving) the message; `timestamp` is the time when the event (`send` or `receive`) occurred; `outcome` denotes `success` or `failure` of the event. In addition to these attributes, we have an attribute referred to as `level` that represents the security level of the tuple. The `level` attribute is assigned by the system and

it cannot be modified by the user.

```
MessageLog(serviceId, msgType, sender, receiver,
                  timestamp, outcome)
```

The queries are expressed using the CQL language [9]. We describe the various types of queries that can be executed at the various tiers.

#### 8.3.1.1 Company Auditing Tier

In the company auditing tier, companies have access only to their own audit records.

In this section we give some sample queries that are executed by *Company1* to detect performance delays and DoS attacks. All the queries are executed at level $[1, \perp]$.

*Query 1 ($Q_1$)*

*Company1* requests service from *CompanyB*. It is trying to check the times when such message could be successfully delivered.

```
SELECT timestamp FROM MessageLog
WHERE msgType = "send" AND outcome = "success"
AND receiver = "CompanyB";
```

*Query 2 ($Q_2$)*

*Company1* requests service from *CompanyB*. It is trying to check the times when such message could not be successfully delivered.

```
SELECT timestamp FROM MessageLog
WHERE msgType = "send" AND outcome = "failure"
AND receiver = "CompanyB";
```

#### 8.3.1.2 Service Auditing Tier

Service auditing tier receives log records from all the companies making use of some service. However, as the queries below demonstrate, not all the queries need to access all the data from the same COI class.

*Query 3 ($Q_3$)*: Level $[\perp, B]$

Log records received at the service auditing tier can be analyzed by the Session Manager to find out whether *CompanyB* is not available for some service.

```
SELECT timestamp FROM MessageLog
WHERE msgType = "send" AND outcome = "failure"
AND receiver = "CompanyB";
```

*Query 4 ($Q_4$)*: Level $[\perp, T]$

Session Manager may wish to find out whether all companies in $COI_2$ are target of some DoS attacks.

```
SELECT timestamp FROM MessageLog
WHERE msgType = "send" AND outcome = "failure"
AND receiver = "CompanyB" OR receiver = "CompanyA" OR
    receiver = "CompanyC";
```

#### 8.3.1.3 Cloud Auditing Tier

Cloud auditing tier gets log records pertaining to all the services. However, the various queries will have different types of security requirements.

*Query 5 ($Q_5$)*: Level $[1, B]$

Cloud Provider may want to look at all records pertaining to `serviceId 5` and measure the delays in order to detect possible man-in-the-middle attack. `serviceId 5` involves *Company1* and *CompanyB*.

```
SELECT  MIN(timestamp), MAX(timestamp)
FROM MessageLog [ROWS 100]
WHERE outcome = "success" AND serviceId = "5";
```

*Query 6 ($Q_6$)*: Level $[1, B]$

Cloud Provider wants to find the delay encountered by *Company1* between sending the request and receiving the service from *CompanyB* for the last 100 tuples.

```
SELECT R.timestamp - S.timestamp  AS  delay
FROM MessageLog R[Rows 100], MessageLog S[Rows 100]
WHERE S.msgType = "send" AND S.outcome = "success"
AND R.msgType = "receive" AND R.outcome = "success"
AND R.receiver = "Company1" AND R.sender = "CompanyB"
AND S.receiver = "CompanyB" AND S.sender = "Company1"
AND S.serviceId = R.serviceId;
```

*Query 7 ($Q_7$)*: Level $[T, T]$

Cloud Provider may want to find out the delay incurred in the different service invocation chains.

```
SELECT MIN(timestamp), MAX(timestamp)
FROM MessageLog[ROWS 100]
WHERE outcome = "success"
GROUP BY serviceId;
```

### 8.3.2 Execution of Cloud Queries

For each tier, we may have one or multiple query processors. In the Company Auditing Tier, we have a single query processor for analyzing each company data. Thus, *Company1* has a single query processor at level $[1, \perp]$. In the Service Auditing Tier, we may have one or more query processors running at different levels. In our examples, we can have a query processor at level $[\perp, B]$ and another one at $[\perp, T]$. Alternatively, we can use $[\perp, T]$ to process both the queries.

Using $[\perp, T]$ to process the query at level $[\perp, B]$ comes at a cost: the query submitted at $[\perp, B]$ must be rewritten such that it can access only those tuples that it is authorized to view. Similarly, for the Cloud Auditing Tier, we may have a single query processor at level $[T, T]$ or two query processors: one at level $[1, B]$ and the other at $[T, T]$.

When a query has been submitted by a user, it must be rewritten to ensure that no unauthorized tuples are returned to the user. Our query rewriting algorithm modifies the algorithm in the following ways. Let $Q_x$ be the original query submitted at level $L(Q_x)$. Let $selectCond(Q_x)$ and $window(Q_x)$ be the selection and window condition associated with the query. The query rewriting algorithm adds a new security conjunct to the existing selection condition. This conjunct ensures that the tuples satisfying the query is dominated by the level of the query. The query rewriting algorithm also restricts the window to filter those tuples that the query is authorized to view; this is denoted by $|window(Q_x)|_{L_{Q_x}}$. The query rewriting algorithm is given below.

**Algorithm 6**: Secure Query Rewriting

**INPUT**: $(Q_x)$

**OUTPUT**: $OPT(Q'_x)$ representing the rewritten query

**if** $window(Q_x) \neq \{\}$ **then**
  $window(Q_x) = |window(Q_x)|_{L(Q_x)}$
**end**
**else**
  $selectCond(Q_x) = selectCond(Q_x) \cup (level \preceq L(Q_x))$
**end**

Let us consider $Q_5$ once again that is submitted at Level $[1, B]$.

```
SELECT MIN(timestamp), MAX(timestamp)
FROM MessageLog [ROWS 100]
WHERE outcome = "success" AND serviceId = "5";
```

If this query is executed by the query processor at Level $[1, B]$, no rewriting is needed. How-

ever, if the query is executed at Level $[T, T]$, the query must be rewritten to ensure that it can view only authorized information. In such a case, the query is rewritten as follows. Note that the DOMINATED BY is similar to keyword *level in* in the MLS language.

```
SELECT MIN(timestamp), MAX(timestamp) FROM
MessageLog [ROWS 100 WHERE level DOMINATED BY [1,B]]
WHERE outcome = "success" AND serviceId = "5"
```

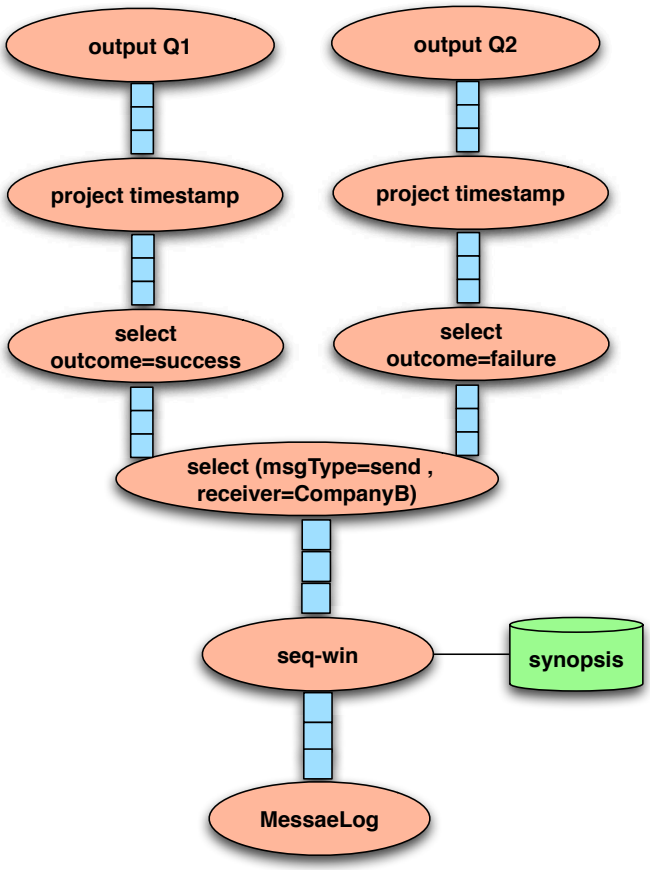


Figure 8.3: Merged Operator Trees of $\mathtt{Q}_1$ and $\mathtt{Q}_2$

With the defined information flow model, we can reuse the ideas of query sharing mechanisms described in replicated MLS-DSMS. Figure 8.3 is an example sharing $Q_1$ and $Q_2$ in stream audit application.

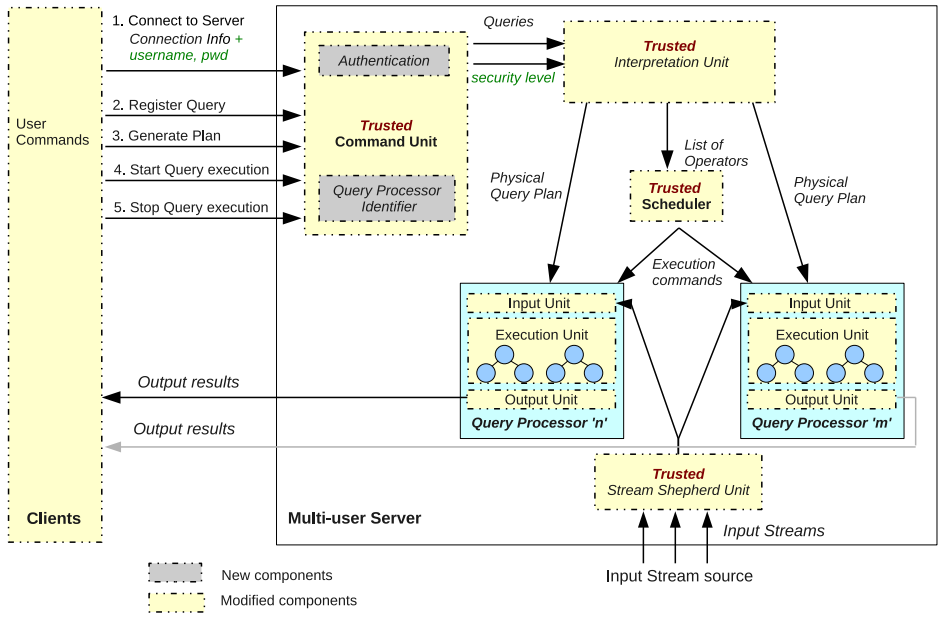


Figure 8.4: CW-DSMS Prototype Architecture

## 8.4 Prototype Implementation

We have developed the replicated CW-DSMS shown in Figure 8.4. This system is a modified version of Replicated MLS-DSMS from our previous work [5, 6]. The CW-DSMS supports: (1) multi-user server with user authentication, (2) replicated query processors executing at different security levels, (3) a global trusted scheduler that schedules operators across all query processors, (4) a global trusted interpretation unit that supports centralized query plan generation for all query processors, (5) trusted stream shepherd operator that takes trusted streams and outputs streams based on the security level of the query processor, (6) security level aware windows, (7) security level aware query operators i.e., modification to blocking operators (e.g., *join*, *average*) to create output tuples with appropriate level identification, and (8) single security level input streams and tuples.

Compare with the replicated MLS-DSMS, we've made changes in the *execution unit* contains

the physical operators and their corresponding algorithms. We have modified the window processing so that it can support audit level models. Besides, we have modified the aggregate and join operator algorithms to compute the least upper bound (LUB) of the input tuples specially for this model and use that as the security level of the output tuples. For example, if an aggregate operation computes the maximum timestamp of three input tuples in levels $[\perp, A], [\perp, B]$ and $[\perp, C]$, the output tuple is in level $[\perp, T]$. On the other hand, all the operators are untrusted. The execution unit accepts the commands from the trusted scheduler and executes the corresponding operators. There is only one operator running at any point of time, since we have only one scheduler. The *output unit* sends the results back to users continuously.

**Theorem 1.** *The proposed architecture enforces the information flow constraints.*

*Proof.* Let $Q$ be a query submitted by a process at level $l$ that operates on the relations and streams in the DSMS. For each stream accessed by the query, the query rewriting operator takes into account the security level of the query and only provides the projection of the respective stream that the process is authorized to view. The query is then forwarded to the processor that executes in the same level as the query.

The query processor at level $l$ can view only those input tuples whose levels are dominated by $l$ and produce output streams at level $l$. Thus, during query processing overt information flow can only occur from levels dominated by level $l$ to level $l$. In the proposed architecture, level $l$ receives tuples from dominated levels and stores them at its own level. There is no common storage that is shared across security levels. Thus, a dominating level cannot manipulate the common storage to pass information to the dominated level. This ensures that there are no covert storage channels. The query processor at level $l$ executes queries only in its allotted time slot as decided by the trusted

scheduler which ensures that there are no timing channels. □

The above claim holds only when we consider the architecture in isolation. However, in real world this is never the case and it is possible for the underlying framework to have covert channels. For example, if the query processors at different levels are executing on the same server it is possible to have storage and/or timing channels.

# Chapter 9

# Prototype Implementation and Experimental Evaluation

In order to apply our MLS-CQL model, query sharing algorithms, and scheduling strategies, we need to make changes in the original DSMS. The vanilla system is based on a client-server architecture, where clients register streams and queries and submit input streams, and the server runs queries over the input data streams. The system maintains a one-to-one relationship between clients and servers. The clients send connection establishment requests, and command messages for registering streams, queries, generate query plans, execute, terminate, etc., to the server. The clients allow users to create queries, create streams, and view outputs, and they also provide input streams to the server. The server generates query plans consisting of operators, queues, and synopses, instantiates queries, schedules and executes queries, provide run-time monitoring, and maintains accuracy and QoS.

## 9.1 Prototype Implementation

In this section, we explain the key components need to be extended or re-designed for addressing MLS system requirements.

### 9.1.1 MLS-CQL Syntax

Our work supports stream-to-stream window operators in MLS-CQL. In the first step, we need to revise the DSMS parser for supporting query syntax such as [Rows 100 level in {U,C}]. For simplicity, we use number 1, 2, 3, and 4 to represent level U, C, S, and TS. So the window [Rows 100 level in {U,C}] can be represented as [Rows 100 12].

Stanford STREAM DSMS uses Bison parser to handle CQL raw queries. We gave a brief summary on the parser processes in STREAM system, more details can be found at [34, 57].

1. All *tokens* (keywords in CQL queries) allowed in CQL statements are defined in source files parser.h and parser.cc. Those input keywords like rows, range, etc are assigned with *symbol numbers* used for further parsing. In Bison the *yytranslate* table is used to map lexical token with symbol numbers.

2. Action (59 terminal symbols like CQL keywords) and GOTO (29 non-terminal symbols like intermediate reducing words) tables are created before parsing. The two tables are abstract concepts. Action table represents the terminated reduction rules and Goto table shows the current state redirection when a keyword is scanned during parsing. There are 157 possible states (from 0 to 156) and 81 reducing rules (in a file parser.yy).

3. In implementation, state change and reduction rules are stored in a couple of tables. Here we only list those needed for MLS revise.

   (a) *yydefact* table stores the default reducing rules (In particular state, the reduction rules are mostly used for all possible valid input symbols).

   (b) *yypact* table tells whether change state or use reduction for current state.

(c) *yytable* table stores the next state or non-default reduction rules for current state indexed by *yypact* table.

(d) *yycheck* table is used for various checking.

Now we explain how parser works. In any moment, parser is in certain state *x*. When a new valid input token (with symbol number *A*) comes, parser first checks *yydefact[x]* table to see if any reduction rule should be apply. If the value is 0, that means no reduction rule is used. Then system checks *yypact[x]* table which determines action/goto entry for current state *x* with look-ahead token *A*. If *yypact[x]* = YYPACT_NINF (always minus value), parser will uses the corresponding reduction rule in *yydefact*. If *yypact[x] = k* positive number, *k* will added to *A* and directed to *yytable* for checking next proper action except using the default reduction rules. Meanwhile, *yycheck[k+A] = A* should be satisfied before working with *yytable*.

After that, we look at the result of *yytable[k+A]*. If it equals to positive number *y*, it means *x* should be changed to next state y with shifting *A* into stack; Otherwise, a negative value *(-z)* means applying reduction rule z (except the one stored in *yydefact*). In the following we first show how to create the new parser rule for [Rows size level]. The following table shows parsing rules for [Rows size] in STREAM system.

**Rows X Level**

Table 9.1: STREAM Parsing Rules for [Rows Size]

| | RW_ROWS(24) | T_INT(44) |
|---|---|---|
| state X | Switch To Y(106) | |
| state Y | Switch To Z(127) | |
| state Z | | Reduce Rule 53 |

To add “level” to the Rows window, we add a new state (A) 157 with revised reducing rule 53:

RW_ROWS N_INT N_INT (revised in parser.yy). The parsing rules should be revised as following:

Table 9.2: MLS-DSMS Parsing Rules for [Rows Size Level]

| | RW_ROWS(24) | T_INT(44) |
|---|---|---|
| state X | Switch To Y(106) | |
| state Y | Switch To A(157) | |
| state A | Switch To Z(127) | |
| state Z | | Reduce Rule 53 |

And we need to revise the four tables accordingly (in file parse.cc).

1. In state X, we don't have changes.

2. In state Y(106), *yypact[106]* = 104, then *yytable[104+44]* = 157 (instead of Z=127).

3. In state A(157), *yydefact[157]* = 0 (no applicable rule), *yypact[157]* = 133, *yytable[133+44]* = 127 (yytable has 178 entries), *ycheck[133+44]* = 44.

4. In state Z(127), *yydefact[127]* = 53 (using reduction rule 53), *yystos[157]* = 44 (internal symbol table).

**Range Unbounded Level**

In trusted MLS-DSMS, we add level filtering in unbounded window during query rewriting. We introduce a new state 158 in order to revise the [Range Unbounded] to [Range Unbounded Level].

Table 9.3: STREAM Parsing Rules for [Range Unbounded]

| | RW_RANGE(25) | UNBOUNDED(28) |
|---|---|---|
| state X(87) | Switch To Y(107) | |
| state Y(107) | | Switch To Z(128) |
| state Z(128) | | Reduce Rule 54 |

the revised reduction rule is 54: RW_RANGE UNBOUNDED N_INT (revised in parser.yy). The parsing table should be revised as following:

Table 9.4: MLS-DSMS Parsing Rules for [Range Unbounded Level]

| | RW_RANGE(25) | UNBOUNDED(28) | T_INT |
|---|---|---|---|
| state X(87) | Switch To Y(107) | | |
| state Y(107) | | Switch To B(158) | |
| state B(158) | | | Switch To Z(128) |
| state Z(128) | | | Reduce Rule 54 |

And we need to revise the four tables accordingly (in file parse.cc).

1. In state X, we don't have changes.

2. In state Y(107), *yypact[107]* = 26, then *yytable[26+28]* = 158 (instead of Z=54).

3. In state B(158), *yydefact[158]* = 0 (no applicable rule), *yypact[158]* = 134, *yytable[134+44]* = 128 (yytable has 179 entries), *ycheck[134+44]* = 44.

4. In state Z(128), *yydefact[128]* = 54 (using reduction rule 54), *ystos[158]* = 44 (internal symbol table).

**Range X Second Level**

Similarly, we need to make parsing rule changes in time-based window with certain time period. Here we show one example changes [Range size second] to [Range size second level]. The original reduction rule is 57, Y(144) = 57.

Table 9.5: STREAM Parsing Rules for [Range X Second]

| | RW_RANGE(25) | SECOND(29) | T_INT(44) |
|---|---|---|---|
| state Y(129) | | Switch To Z(144) | |
| state Z(144) | | | Reduce Rule 57 |

Here we show the revised table. A new state 159 is introduced.

Table 9.6: MLS-DSMS Parsing Rules for [Range X Second Level]

| | RW_RANGE(25) | SECOND(29) | T_INT(44) |
|---|---|---|---|
| state Y(129) | | Switch To C(159) | |
| state C(159) | | | Switch To Z(144) |
| state Z(144) | | | Reduce Rule 57 |

And we need to revise the four tables accordingly.

1. In state Y(129), *yypact[129]* = 102, then *yytable[102+29]* = 159 (instead of 57).

2. In state C(159), *yydefact[159]* = 0 (no applicable rule), *yypact[159]* = 135, *yytable[135+44]* = 144 (yytable has 180 entries), *yycheck[135+44]* = 44.

3. In state Z(144), *yydefact[144]* = 57 (using reduction rule 57), *yystos[159]* = 44 (internal symbol table).

### 9.1.2 Sharing Plan Generation

In Chapter 3 we show the intermediate products of system from CQL raw queries to execution units during query plan generation.

```
CQL query -> Syntactic nodes in parse tree -> Semantic objects
-> Logical operation (logical plan) tree
-> Physical operation (physical plan) tree -> Execution units
```

First we need to make a choice in which step to perform sharing analysis. We would like to do it before physical plan generation, the earlier the better. In general we can run sharing analysis while the system performs one of three steps:

1. System is accepting raw input queries. There are two difficulties in this choice: the first one is the complexity of table aliases. Suppose there are two queries as following:

```
Qa: Select M.id From Vitals M, Positions N;

Qb: Select N.id From Vitals N, Positions M;
```

They are identical if Vitals and Positions are pointed to the same input streams. However the alias complicates the analysis.

The second problem is identifying the type of input: relation or stream. Vitals in $Q_a$ and $Q_b$ can be a fixed size table, or a continuous input stream. From the raw input the system cannot tell the type of inputs. So sharing analysis in raw query is not applicable.

2. In syntactic parser trees. CQL raw queries are constructed as a list of nodes. These syntactic nodes only contain the information after parsing, like the attributes, operations, window size, etc. They are not connected with the input streams/tables; So syntactic nodes are inappropriate to use for sharing analysis.

3. In Semantic objects. Those objects are well-structured and connected with inputs. For example, a CQL query can be one of two basic types such as SFW (Select-From-Where) and Binary-join. In SFW, there are three blocks select, from and where, and each of them contains nodes. For nodes in from block, it must specify the input source, input window kind and size, etc. All nodes are linked and ready to generate a logical plan.

   We perform the sharing analysis by comparing existing semantic objects with the arriving semantic objects from the new query. Since sharing starts from bottom to top, we first compare the input source in FROM clause, then the WHERE, and the SELECT clause. The analysis tries to share nodes as many as possible in the operator tree. According to the sharing position (FROM, WHERE or WHERE) and type (partial or complete), the new

query is able to reuse the logical and physical plans from existing queries before generating its own plan.

We need to set up buffers for storing the semantic objects, logical and physical plans of all executing plans. In sharing mechanism, if a query stops, the system checks whether there are some executing queries reusing its plan. In this case, the query plan will stay for others even though the query is terminated.

### 9.1.3 Execution/Generation Time Measuring

In STREAM prototype, there are two additional operators *stream-source* and *output* to be created for every query plan of CQL queries. The stream-source operator connects input streams and produce tuples only with positive sign. It acts like the stream source producer and located the bottom most node in the operator tree. Output operator is the top most node in operator tree, which constructs the output in certain form to send it back to the users.

Our experiments require keeping track of the process time for certain query against limited size inputs. We are only interest in the execution time of the plain operator tree without the two additional operators. So the timer starts from the first tuple enters to an operator next to stream-source, ends when the last tuple goes out from output operator. We've made the changes to add the timer function.

Query plan generation time is most straightforward. Once the user decides to run queries, no more queries are accepted by this user. So the generation time can be calculated between the start of first plan generation and the end of the last plan.

### 9.1.4 LUB Level Computation

Least Upper Bound (LUB) computation for level attribute is critical for blocking operators like aggregations and join in our MLS-DSMS. Join LUB is relatively simple. Suppose there is a join operator in S level which joins two tuples $t_1$, $t_2$ from C and U respectively. The output joined tuple should be in level C by performing LUB($t_1$, $t_2$). A small piece of code is used in our prototype:

```
outputElement.level =
    ((t1.Level>= t2.level)?t1.level:t2.level);
```

Aggregation operators like sum, count, max, min, avg are different from join. Every time a new tuple arrives at or an old tuple expires from the computation window, there will be an aggregation computation. It is time and resource consuming if we preform LUB of all involved data when a computation happens. Instead in our system, we keep count numbers for each level from tuples in the computation window. The size of counts are bound to the size of window, as well as the computation is only increase (for new input) or decrease (for expired buffered data) in most cases. Specifically, we use the Bitwise operator to improve the performance:

```
if(inputElement.sign == plus)
   aggrLUBLevel[1111 & inputElement.level]++;
else
   aggrLUBLevel[1111 & inputElement.level]--;
```

$aggrLUBLevel$ is an array storing the level counts. Index 0 for the array is to store the LUB level of aggregation results. Since we use positive number 1,2,3,4 to represent level U,C,S,TS, binary operations return the actual index in the array for count number updates. After that, we will check if we need to update the output LUB level of aggregation result. The following piece of code shows level update is triggered only if the expired tuple equals to current LUB level of output as well as the count for the expired tuple is 0.

```
if(inputElement.sign == negative &&
   inputElement.level == aggrLUBLevel[0] &&
   aggrLUBLevel[inputElement.level] == 0){
       for(int k=aggrLUBLevel[0]-1; k>=1; k--){
          if(aggrLUBLevel[k]!=0){
              aggrLUBLevel[0] = k;
              break;
          }
       }
}
```

When the update happens, we try to find the count number not equal to 0 from highest (4) to lowest (1). Then we assign the level with non-zero count to the LUB level. With the special care for LUB levels in blocking operators, the output result can be corrected labelled and reused safely via sharing mechanisms.

### 9.1.5 Scheduling Method

In the vanilla STREAM prototype, the scheduler uses round-robin algorithm to schedule operators. We have modified the scheduler so that it can handle scheduling of queries in more than one query processor. The scheduler maintains all executing query plan information shared by the trusted interpretation unit. When a query plan is received by the scheduler, operators in the plan are scheduled for execution from bottom to top order. The scheduler sends out commands (including plan id and operator id) to the appropriate query processor to start executing an operator. The operators execute at least once per scheduling round. When a new plan arrives at the scheduler, operators of that plan will be scheduled in the next execution round. Such mechanism prevents starvation of late coming queries, as each operator is scheduled every round. In every round, each operator processes a maximum of 170 data tuples before switching to other operators. The DSMS

can process a maximum input 100,000 tuples per second. “First come first serve” strategy is used for executing the query plans. We adapt the round-robin method in our trusted scheduler which is able to schedule operators across all query processors.

So the original round-robin does not assign time slice for operators but maximum processing tuples for each round. In our time-slot algorithm, we set up 250 million-second for each level since the heart beat is 1 second. In each time slot, we run operators in multiple rounds in specific levels. To ensure the fast output, we adapt the ideas of handling maximum 170 tuples for each operator in each round, while the scheduler will run it as many as possible during the assigned time slot.

## 9.2 Experiment Setup

MLS DSMSs are developed to achieve two main goals: security protection enforcement by introducing trusted components and safe scheduling strategies, and performance improvement via sharing queries in the same or different levels. We conduct empirical experiments to evaluate the overhead of the security mechanisms and the performance gain of sharing ability between normal and MLS-DSMS.

There are some metrics that can be used to study the performance and overhead of multi-level security. Some of the most common metrics are: tuple latency, throughput, result accuracy, starvation, number and complexity of trusted components, storage requirements, etc. In DSMS applications, QoS is the most critical factor to evaluate the system performance. So we use *response time* for query plan generation and execution to investigate the pros and cons of MLS DSMSs. For the experiments, we are planning to generate synthetic data with level labels and using different kinds of queries like select, project, join, and aggregation (e.g., average). In general, we aim to find answers of the following questions:

- MLS Architectures vs. Vanilla System: What is the overhead caused by introducing the multilevel security processing components.
- Modified scheduling overhead: Since scheduling has to be modified to incorporate security levels, what are the impacts on query processing?
- Sharing vs. No Sharing: What are the effects of sharing and no sharing of queries? This involves comparing all the approaches of complete and partial sharing.
- Sharing in the same vs. across different levels: What are the benefits of sharing queries across different levels?
- In CW-DSMS, what is the overhead causing by introduction of trusted components?

In the following sections, we first provide the details of experiment setup, then present the experimental evaluations on those secure DSMSs.

**Environment**: All the experiments were conducted in a standalone system with Intel i7 Q820 1.73GHZ Quad core Processor, 6GB RAM, and Ubuntu 11.10 64bit OS. Processes except DSMS are shut down and there is no internet/bluetooth/wireless connection to the machine.

**Inputs:** Experiments are under the simulation scenario of the battlefield monitoring application discussed in Chapter 4. Each soldier equips with sensors sending out vital and position info to the control center with DSMS continuously. The two input stream schemes are showed as following, and Each tuple is associated with a security level, which can be TS, S, C, or U, where U $<$ C $<$ S $<$ TS.

```
Vitals(soldier id (sid), blood pressure (bp), pulse rate (pr),
weight (weight), level (level));
```

```
Positions(soldier id (sid), latitude (lat), longitude (lon),
level (level));
```

For each of two *Vitals* and *Positions* input streams, we set up separate input files containing numbers of tuple in different sizes like 500 thousand, 1 million, 2 million and 4 million. According to experiment needs, the content of input files can be different. For example, all tuples of an input can be in the same level, or the number of tuples in each level is 1/4 of total tuples. To create light/heavy load situation, the input data rates vary from 10,000 to 100,000 tuples per second. Besides, we are executing different kinds of queries as simple select, aggregation, join, as well as mixed types of queries.

**Data Collection:** Our goal is to compare the response time of plan generation and query executions between different DSMSs or approaches. To record the exact running time, we should eliminate other time factors introduced by irrelevant stages like query registration, plan analysis, operate tree generation, returning results, etc. So a timer is set up to keep track of exact response time. After plan generation, each registered query is transformed to an operator tree ready for execution. The timer begins to work when first input tuple enters the bottom most operator (window) of the tree, and stops when the last tuple exits from the top operator. The duration is the pure response time of query execution. Similarly, we use the timer to keep track of the plan generation time.

For each experiment, we will run five times and discard the first two runs. The average execution time from the last three runs and the standard deviation will be presented as outcomes. We will compare the overheads and performance gains in MLS systems with the benchmarks of non-security control and no sharing systems respectively.

## 9.3 Experiments on Replicated MLS-DSMS

The followed results are based on experiments of the vanilla STREAM system and the replicated MLS-DSMS prototype.

### 9.3.1 Experiment Expectations

Through the experiments, we should be able to know the following facts of MLS-DSMS implementations:

1. The overhead of MLS components. It should be insignificant to execution time otherwise the MLS DSMS is not useful for real-world applications.

2. Performance improvement by sharing approach. Sharing between queries reduce the complexities of plan generation as well as query execution by reusing the existing operator tree structures and computation results. Specially, complete sharing should benefit more than partial sharing.

3. The overhead of MLS scheduling strategies used in replicated MLS-DSMS. For specific scheduling approach, the overhead is determined by the system load, data distribution and query number in each level. Running the same set of queries under different parameters, we can see which scenario is (or not) suitable for particular scheduling approach.

### 9.3.2 Overhead of MLS processing

We used three different data sets with 1, 2, and 4 Million tuples. The data input rate is 20,000 tuples per second. In each set, the number of tuples in each level was 1/4 of total tuples. In order to

detect the overhead cost only, both vanilla and replicated MLS DSMS use the default round-robin scheduling strategy with no load shedding.

Table 9.7: Performance Overhead Due to MLS Processing

| | Data Size (tuples) | Average Execution Time (ms) | | Overhead Due to MLS-DSMS (in %) | Standard Deviation (ms) | |
|---|---|---|---|---|---|---|
| | | Vanilla | MLS-DSMS | | Vanilla | MLS-DSMS |
| Exp 1 | 1M | 50063 | 50068 | 0.010% | 5.29 | 4.58 |
| | 2M | 100069 | 100079 | 0.011% | 9.29 | 4.04 |
| | 4M | 200071 | 200082 | 0.005% | 2.52 | 2.52 |
| Exp 2 | 1M | 50065 | 50070 | 0.011% | 4.73 | 2.31 |
| | 2M | 100068 | 100077 | 0.009% | 7.77 | 6.08 |
| | 4M | 200075 | 200081 | 0.003% | 3.51 | 7.21 |
| Exp 3 | 1M | 50528 | 50596 | 0.134% | 12.50 | 45.54 |
| | 2M | 100535 | 100628 | 0.093% | 5.86 | 25.66 |
| | 4M | 200557 | 200655 | 0.049% | 32.50 | 79.05 |
| Exp 4 | 1M | 53072 | 53544 | 0.882% | 131.87 | 183.44 |
| | 2M | 103850 | 104456 | 0.580% | 397.64 | 441.46 |
| | 4M | 204463 | 205576 | 0.541% | 243.36 | 210.66 |

We used the four experiments to study the performance overhead. The average execution time are shown in Table 9.7. Experiment 1, 2 and 3 are running one simple select, one average with group by, and one join queries respectively in both vanilla and replicated MLS-DSMS in highest level. As join is an expensive operation when compared to other operations, the time by experiment 3 is more when compared to experiments 1 and 2. In general the overhead due to MLS processing is negligible under all data sets as shown in Table 9.7.

In experiment 4, there are six queries that included multiple copies of the same queries used in the previous experiments. Two copies of each query are used in experiment 1, 2, and 3, respectively. These six queries used the same input streams. As shown in Table 9.7, the overhead due to multilevel processing is between 0.54% and 0.88%. The system takes less performance hit with 4M tuples when compared to 1M tuples as the system stabilize over long run.

### 9.3.3 Overhead of MLS Scheduling Strategy

In experiment 5, we study the overhead caused by the implementation of the fixed time slot round-robin scheduler vs. the regular round robin scheduler. We executed four queries, each running at different security levels. The input streams (Vitals1, Vitals2, Vitals3, and Vitals4) are replications of the original Vitals stream. The input streams contain only unclassified tuples. We used a input rate of 40,000 tuple/sec and data sets of 1M, 2M, and 4M tuples. The queries are shown below.

```
Q1(TS): SELECT sid, weight
        FROM Vitals1[Rows 100 level in {TS,S,C,U}];

Q2(S):  SELECT sid, weight
        FROM Vitals2[Rows 100 level in {S,C,U}];

Q3(C):  SELECT sid, weight
        FROM Vitals3[Rows 100 level in {C,U}];

Q4(U):  SELECT sid, weight
        FROM Vitals4[Rows 100 level in {U}];
```

Table 9.8: Overhead Due to Trusted Scheduler and Stream Shepherd Operator

| | Data Size (tuples) | Average Execution Time (ms) | | Overhead (in %) | Standard Deviation (ms) | |
|---|---|---|---|---|---|---|
| | | MLS-DSMS Regular Scheduler | MLS-DSMS MLS Scheduler | | MLS-DSMS Regular Scheduler | MLS-DSMS MLS Scheduler |
| Exp 5 | 1M | 26928 | 27424 | 1.807% | 74.22 | 60.93 |
| | 2M | 51853 | 52318 | 0.889% | 58.80 | 58.03 |
| | 4M | 101966 | 102541 | 0.560% | 67.87 | 35.04 |

As shown in Table 9.8, the overhead due to the time slot scheduling is between 0.56% and 1.80%. The result is good because there are same input load in each time slot (one query per time slot). In this case we can see this scheduling method is under suitable scenario. On the other hand, if one of the levels has more tuples, then the overhead might be higher as other time slots have to be exhausted before that level is rescheduled.

### 9.3.4 MLS Query Sharing

Three input data sets with 500 Thousand, 1 Million, and 2 Million tuples were used at an input rate of 20,000 to 40,000 tuples per second. In each input stream, the number of tuples in each level was 1/4 of total tuples. In order to detect the pure sharing gain, in both no sharing and sharing replicated MLS systems we used the default round-robin scheduling strategy without load shedding.

We are using p value of T-test to evaluate the significance of differences by sharing. The T-test is a statistical hypothesis test to determine if two data sets are significantly different from each other. We use two tailed assuming unequal variances as setup with confidence level 0.05. For the two sets of data for comparison, if the p value is smaller than 0.05, we say the sharing difference is significant.

## Complete Sharing

Four experiment sets are conducted to study the performance gain due to complete sharing. We measured time costs of query execution and plan generation.

Table 9.9: Complete Sharing Execution - Performance Gain

| | Data Size (tuples) | Average Execution Time (ms) | | Performance Gain (in %) | Standard Deviation (ms) | | p-value |
|---|---|---|---|---|---|---|---|
| | | Complete Sharing | No-Sharing | | Complete Sharing | No-Sharing | |
| Exp 1 | 500K | 17795.0 | 19228.0 | 8.053% | 524.9 | 467.7 | 2.7E-02 |
| | 1M | 34189.3 | 37321.0 | 9.160% | 1173.5 | 1038.2 | 2.6E-02 |
| | 2M | 68207.7 | 72673.3 | 6.547% | 995.7 | 577.9 | 5.4E-03 |
| Exp 2 | 500K | 45443.7 | 50023.7 | 10.078% | 323.7 | 790.4 | 4.3E-03 |
| | 1M | 87061.3 | 95074.7 | 9.204% | 191.8 | 498.9 | 3.4E-04 |
| | 2M | 175681.7 | 193476.0 | 10.129% | 445.4 | 1802.6 | 2.2E-03 |
| Exp 3 | 500K | 30386.3 | 33563.0 | 10.454% | 367.4 | 503.4 | 1.3E-03 |
| | 1M | 54479.0 | 65988.0 | 21.126% | 543.5 | 1113.2 | 6.3E-04 |
| | 2M | 105859.7 | 131491.3 | 24.213% | 1156.7 | 2496.3 | 7.3E-04 |
| Exp 4 | 500K | 34697.0 | 36125.3 | 4.116% | 151.1 | 282.6 | 4.2E-03 |
| | 1M | 67581.3 | 70112.0 | 3.745% | 330.1 | 324.5 | 7.0E-04 |
| | 2M | 135166.3 | 139601.3 | 3.281% | 1033.0 | 788.8 | 5.1E-03 |

Table 9.10: Complete Sharing Plan Generation - Performance Gain

| | Average Execution Time for 3 Runs (ms) | | Improvement due to sharing (in %) | Standard Deviation between 3 Runs (ms) | |
|---|---|---|---|---|---|
| | Complete Sharing | No Sharing | | Complete Sharing | No Sharing |
| Exp 1 (Plan Generation) | 762.0 | 774.0 | 1.575% | 2.7 | 6.0 |
| Exp 2 (Plan Generation) | 767.6 | 776.4 | 1.158% | 3.5 | 8.8 |
| Exp 3 (Plan Generation) | 374.2 | 378.7 | 1.188% | 3.7 | 7.0 |
| Exp 4 (Plan Generation) | 771.6 | 784.0 | 1.613% | 3.2 | 5.4 |

Experiment 1, 2 and 3 are running identical queries as nine select, nine average with group-by, and five join respectively in both sharing and no sharing MLS DSMS in highest level. Experiment 4 is mixed nine queries contains three select, three average with group-by, and three join from the three previous experiments. As shown in Table 9.9 under Exp 3, the highest performance gain for 5 join queries was between 10.45% and 24.21%. Besides, the execution time performance gain of the 4 experiments was between 3.28% to 24.21%. This variation is mainly due to the processing time took by operators and due to the change in input rate. As load shedding is not enabled, we can fine tuned the input rates to avoid inconsistent results. For instance, if the input rate is increased to 100,000 tuples per second, the DSMS was producing inconsistent results over the five runs of the same experiment.

On the other hand, p values of all experiments are smaller than 0.05 so the performance differences are significant.

As shown in Table 9.10 the performance gain due to not creating already existing plans was between 1.15% and 1.61%. But the gain is negligible when the standard deviation over the runs is taken into account. There is not a lot of performance gain as sharing analysis consumes resources.

## Partial Sharing

Two experiments were set up to enable partial sharing. Experiment 1 and 2 are conducted with five join and seven average queries respectively. The From and Where clauses were identical (to make sure the same input data) in all the queries and the where clause in select operator was different (e.g., different numbers of attributes or different aggregation computation types).

Table 9.11: Partial Sharing Execution - Performance Gain

| | Data Size (tuples) | Average Execution Time (ms) | | Performance Gain (in %) | Standard Deviation (ms) | | p-value |
|---|---|---|---|---|---|---|---|
| | | Partial Sharing | No-Sharing | | Partial Sharing | No-Sharing | |
| Exp 1 | 500K | 18770.3 | 19970.3 | 6.393% | 428.1 | 148.2 | 2.9E-02 |
| | 1M | 35383.0 | 36733.7 | 3.817% | 436.0 | 328.9 | 1.5E-02 |
| | 2M | 69903.0 | 72841.7 | 4.204% | 527.6 | 722.8 | 6.1E-03 |
| Exp 2 | 500K | 33588.3 | 35839.0 | 6.701% | 437.5 | 333.2 | 2.7E-03 |
| | 1M | 67405.0 | 72322.0 | 7.295% | 653.9 | 944.0 | 2.8E-03 |
| | 2M | 132964.7 | 140305.0 | 5.520% | 1007.9 | 1415.5 | 2.7E-03 |

Table 9.12: Partial Sharing Plan Generation - Performance Gain

| | Average Execution Time for 3 Runs (ms) | | Improvement due to sharing (in %) | Standard Deviation between 3 Runs (ms) | |
|---|---|---|---|---|---|
| | Partial Sharing | No Sharing | | Partial Sharing | No Sharing |
| Exp 1 (Plan Generation) | 375.0 | 377.9 | 0.770% | 3.3 | 4.7 |
| Exp 2 (Plan Generation) | 572.4 | 578.7 | 1.087% | 3.1 | 2.3 |

Based on Table 9.11, the performance gain due to partial sharing was between 3.81% and 7.29%. On the other hand, the performance gain due to plan generation was 1% or less as shown in Table 9.12. This shows that analyzing the existing plans for partial sharing does not cause overhead in the system. Similar to experiments on complete sharing, p values of all experiments are smaller than 0.05 so the performance differences are significant.

From the results on replicated architecture, we can see the overhead by MLS components is

insignificant and sharing mechanism provides better QoS by reducing the response time. On the other hand, overhead of the new scheduling strategy like fixed time slot is not big if running in certain suitable scenario.

## 9.4 Experiments on Trusted MLS-DSMS

### 9.4.1 Experiment Expectations

In this section, we discuss the experimental evaluations conducted to study the overhead of trusted query rewriter, as well as performance gain by sharing queries across different levels.

1. The overhead of MLS components in trusted system. It should be insignificant to execution time. We are running experiments for comparison between vanilla DSMS and no-sharing trusted MLS-DSMS.

2. Performance improvement by sharing approach. We are running the experiments between replicated and trusted systems. We are running the experiments in special scenarios where there are overwhelming number of queries in certain same level arrives, as well as sharing across different levels in trusted MLS-DSMS vs. non-sharing in replicated structure.

Three input data sets with 500 Thousand, 1 Million, and 2 Million tuples were used at an input rate at 20,000 to 40,000 tuples per second. Tuples contained a security level ($TS > S > C > U$). In each set, the number of tuples in each level was 1/4 of total tuples.

### 9.4.2 Vanilla DSMS Vs. No-sharing Trusted MLS-DSMS

In order to study the overhead when compared to Vanilla DSMS due to the secure rewriting module, we disabled the sharing ability of Trusted MLS-DSMS. We ran 3 experiments with input rate 20000

tuples per second to 1) measure the plan generation time, and 2) measure the query execution time (the time taken from first tuple entering the S2R operator and last tuple exiting the query).

For trusted MLS-DSMS, all the queries are executed at the TS security level. The reason is that Vanilla system does not classify users or queries based on security levels.

1. Experiment 1 (Range Unbounded Queries): We ran 4 identical queries. Note that the queries run in trusted MLS-DSMS are rewritten in different form.

```
Vanilla: SELECT sid, weight FROM Vitals;

Trusted: SELECT sid, weight
         FROM Vitals[Range Unbounded level in {TS,S,C,U}];
```

From Table 9.13, the performance overhead during execution is between 0.012% and 0.031% and the overhead during plan generation is from 3.876% as shown in Table 9.14.

Table 9.13: Vanilla DSMS Vs. No-sharing Trusted MLS-DSMS : Execution

| Execution Input Rate 20000 | Data Size (tuples) | Average Execution Time (ms) | | Overhead (in %) | Standard Deviation (ms) | |
|---|---|---|---|---|---|---|
| | | Vanilla | No-sharing Trusted | | Vanilla | No-sharing Trusted |
| Exp 1 | 500K | 25051 | 25058 | 0.031% | 3.79 | 2.08 |
| | 1M | 50062 | 50073 | 0.022% | 2.65 | 2.00 |
| | 2M | 100072 | 100084 | 0.012% | 4.04 | 5.03 |
| Exp 2 | 500K | 26967 | 27419 | 1.677% | 96.01 | 201.64 |
| | 1M | 51104 | 51861 | 1.481% | 70.06 | 93.58 |
| | 2M | 101404 | 102104 | 0.691% | 44.09 | 94.14 |
| Exp 3 | 500K | 35016 | 35047 | 0.089% | 2.52 | 2.65 |
| | 1M | 60056 | 60083 | 0.046% | 1.73 | 3.21 |
| | 2M | 110084 | 110111 | 0.024% | 2.65 | 2.08 |

2. Experiment 2 (JOIN Queries): We ran 4 identical queries.

```
Vanilla: SELECT *
         FROM Vitals[Rows 100], Positions[Rows 100]
         WHERE Vitals.sid = Positions.sid;
```

Table 9.14: Vanilla DSMS Vs. No-sharing Trusted MLS-DSMS : Plan Generation

| Plan Generation | Average Execution Time (ms) for 3 Runs | | Overhead (in %) | Standard Deviation (ms) between 3 Runs | |
|---|---|---|---|---|---|
| | Vanilla | No-sharing Trusted | | Vanilla | No-sharing Trusted |
| Exp 1 | 269.4 | 279.9 | 3.876% | 2.6 | 2.5 |
| Exp 2 | 272.2 | 279.1 | 2.531% | 2.7 | 2.2 |
| Exp 3 | 274.8 | 282.4 | 2.790% | 1.6 | 1.8 |

```
Trusted: SELECT *
         FROM Positions[Rows 100 level in {TS,S,C,U}],
              Vitals[Rows 100 level in {TS,S,C,U}]
         WHERE Vitals.sid = Positions.sid;
```

Overhead of execution is between 0.691% and 1.677% as shown in Table 9.13 and plan generation is 2.531% as shown in Table 9.14.

3. Experiment 3 (Range Window Queries): We ran 4 identical queries.

```
Vanilla: SELECT AVG(bp)
         FROM  Vitals[Range 10 seconds]
         WHERE  bp > 100 AND level in {S};

Trusted: SELECT AVG(bp)
         FROM  Vitals[Range 10 seconds level in {S}]
         WHERE  bp > 100;
```

As shown in Table 9.13, the performance overhead due to operators like Range window and Average is between 0.024% and 0.089%. The rewriting and optimization overhead from Table 9.14 is 2.790%.

As discussed above the overhead caused due to Trusted implementation over Vanilla DSMS is almost negligible during query execution and is under 3.876% during plan generation.

### 9.4.3 Replicated Vs. Trusted

The major differences between our replicated/hybrid and trusted architecture implementations are: (1) replicated system does not have query rewriter module. (2) replicated uses time-slot scheduling which assigns identical time duration for running queries in each level to avoid covert channels. While trusted system uses round-robin scheduler for all queries in different levels. (3) replicated system establishes one server instance for users in each level, while trusted has only one server instance for users from all levels. and (4) replicated has *trusted stream shepherd operator* which can filter unqualified stream inputs for each server instance. Thus, time-based window in replicated does not support level filtering and user can specify requesting levels in the WHERE condition for computation. While trusted time-based window supports level filtering, since the stream shepherd operator does not perform filtering.

To find the execution performance differences between these two systems, we ran two experiments with input rate 40000 tuples per second. We increased the input rate in order to create a heavy load situation so that we can observe how the performance can be improved by sharing queries in the same level and across levels. Besides, we still perform T-test to evaluate the significance of sharing performance.

1. Experiment 4 (Sharing in Same Level): There are 4 users in TS level and each one runs the following Join query:

```
Replicated and Trusted:
 SELECT Vitals.sid, weight, location
 FROM   Positions[Rows 50 level in {U}],
        Vitals[Rows 50 level in {U}]
 WHERE  Vitals.sid = Positions.sid;
```

In replicated system all those queries must be run in the fixed time slot in TS level. While in trusted system, operators are run in the round-robin fashion. From Table 9.15, the performance gain in the trusted system is between 63.659% and 108.909%. The above performance benefit is due to the fact that the CPU is not idle in the trusted architecture.

Table 9.15: Replicated Vs. Trusted : Execution

| Execution Input Rate 40000 | Data Size (tuples) | Average Execution Time | | Performance Gain (in %) | Standard Deviation | | p-value |
|---|---|---|---|---|---|---|---|
| | | Replicated | Trusted | | Replicated | Trusted | |
| Exp 4 | 500K | 29297 | 14024 | 108.909% | 888.99 | 236.16 | 5.7E-04 |
| | 1M | 49032 | 27105 | 80.898% | 1073.54 | 324.86 | 3.1E-04 |
| | 2M | 88957 | 54355 | 63.659% | 1266.34 | 471.32 | 9.6E-05 |
| Exp 5 | 500K | 35899 | 21466 | 67.241% | 766.06 | 567.65 | 2.5E-05 |
| | 1M | 66061 | 40266 | 64.063% | 2058.97 | 943.91 | 4.3E-04 |
| | 2M | 113206 | 70044 | 61.620% | 3530.80 | 1598.39 | 4.7E-04 |

2. Experiment 5 (Sharing across Levels): There are 2 users in TS and S level, respectively. Each one runs all the following 4 Join queries. Note that the four queries cannot be shared because of the different input size in window operators.

```
Replicated and Trusted:
 SELECT Vitals.sid, weight, location
 FROM   Positions[Rows 90 level in {S,C,U}],
        Vitals[Rows 10 level in {S,C,U}]
 WHERE  Vitals.sid = Positions.sid;

 SELECT Vitals.sid, weight, location
 FROM   Positions[Rows 80 level in {S,C,U}],
        Vitals[Rows 20 level in {S,C,U}]
 WHERE  Vitals.sid = Positions.sid;

 SELECT Vitals.sid, weight, location
 FROM   Positions[Rows 70 level in {S,C,U}],
        Vitals[Rows 30 level in {S,C,U}]
 WHERE  Vitals.sid = Positions.sid;

 SELECT Vitals.sid, weight, location
```

```
FROM   Positions[Rows 60 level in {S,C,U}],
Vitals[Rows 40 level in {S,C,U}]
WHERE  Vitals.sid = Positions.sid;
```

In the replicated system only queries issued by the users in the same level can be used for sharing analysis. So queries in level S cannot be shared with queries in level TS even though they are in the similar context. On the other hand, in trusted MLS-DSMS queries can be shared across levels.

From the results in Table 9.15 performance gain due to sharing the queries across levels is between 61.620% to 67.241%.

The p values of the two experiments are smaller than 0.05 so the performance differences are significant.

## 9.5 Experiments on CW-DSMS

We conducted experiments to compare the performance between the old vanilla DSMS and the CW-DSMS prototype. Except experiment 6 (join operation), for all other experiments, we used three different data sets with 2, 5, and 10 million tuples with a data input rate of 50,000 tuples per second. Each tuple is associated with a COI class in terms of [$x$,0] or [0,$y$], where $x$ refers to company 1 or 2 and $y$ can be one of the companies A, B, or C. $0$ means the public knowledge $\perp$. For all experiments, the round robin method is used for operator scheduling. The experiment results are shown in Table 9.16. We measured the query execution time (the time taken from first tuple entering the first operator of the query plan and last tuple exiting the query) for the following experiments.

1. **Experiment 1: Company auditing in level [1, $\perp$]**

Table 9.16: Performance Overhead of Chinese Wall Processing

| | Data Size (tuples) | Average Execution Time (ms) | | Overhead Due to | Standard Deviation (ms) | |
|---|---|---|---|---|---|---|
| | | Vanilla | CW-DSMS | | Vanilla | CW-DSMS |
| Exp 1 | 2M | 40031 | 40041 | 0.025% | 2.52 | 2.52 |
| | 5M | 100046 | 100055 | 0.009% | 2.08 | 3.21 |
| | 10M | 200057 | 200063 | 0.003% | 3.21 | 2.52 |
| Exp 2 | 2M | 40032 | 40039 | 0.017% | 2.52 | 1.53 |
| | 5M | 100042 | 100049 | 0.007% | 2.08 | 2.65 |
| | 10M | 200052 | 200056 | 0.002% | 3.51 | 3.51 |
| Exp 3 | 2M | 40030 | 40041 | 0.027% | 2.89 | 1.73 |
| | 5M | 100043 | 100057 | 0.014% | 3.06 | 3.51 |
| | 10M | 200054 | 200065 | 0.005% | 2.08 | 3.06 |
| Exp 4 | 2M | 40044 | 40053 | 0.023% | 3.51 | 2.65 |
| | 5M | 100056 | 100065 | 0.009% | 2.89 | 4.93 |
| | 10M | 200062 | 200069 | 0.003% | 1.53 | 2.52 |
| Exp 5 | 2M | 40047 | 40059 | 0.030% | 2.52 | 3.51 |
| | 5M | 100082 | 100099 | 0.018% | 4.93 | 6.81 |
| | 10M | 200094 | 200116 | 0.011% | 4.58 | 6.24 |
| Exp 6 | 100K | 40532 | 40621 | 0.218% | 11.36 | 30.35 |
| | 250K | 100593 | 100726 | 0.132% | 9.85 | 30.09 |
| | 500K | 200628 | 200789 | 0.081% | 14.47 | 33.86 |

```
SELECT timestamp FROM MessageLog
WHERE msgType = "send" AND outcome = "success"
AND receiver = "CompanyB";
```

In order to maximize the difference in execution time, we used 100% selectivity (all tuples are in level [1, ⊥]) on both the systems, so that no tuples are filtered by the select operator. As shown in Table 9.7 under Exp 1, the performance overhead due to security modification to the vanilla DSMS is negligible for all the data sets used, and it is between 0.003% and 0.025%.

2. **Experiment 2: Company auditing in level [1, ⊥]** Here we used 50% selectivity. The performance overhead is again negligible, and is between 0.002% and 0.017%.

```
SELECT timestamp FROM MessageLog
WHERE msgType = "send" AND outcome = "failure"
AND receiver = "CompanyB";
```

3. **Experiment 3: Service auditing in level [⊥, B]** In the service auditing experiments 3 and 4, the input stream has tuples at 5 different levels: [1, ⊥], [2, ⊥], [⊥, A], [⊥, B] and [⊥, C]. Tuples in each level occupied 20% of the input stream. Since query 3 runs in level [⊥, B], only 20% tuples from inputs should be processed by query 3. So in the vanilla system we must include the condition based on security level in the query:

```
Vanilla:
 SELECT timestamp FROM MessageLog
 WHERE msgType = "send" AND outcome = "failure"
 AND receiver = "CompanyB" AND level = [0,B];

CW-DSMS [0,B]:
 SELECT timestamp FROM MessageLog
 WHERE msgType = "send" AND outcome = "failure"
 AND receiver = "CompanyB";
```

In CW-DSMS, unqualified tuples i.e., tuples not in level [⊥, B], are filtered by the trusted stream shepherd operator due to the replicated architecture. The performance overhead is between 0.005% and 0.027% for all data sets used, which is again negligible.

4. **Experiment 4: Service auditing in level [⊥, T]** Using the same input from experiment 3, the selectivity becomes 60% because level [⊥, T] is authorized to access inputs with levels [⊥, A], [⊥, B] and [⊥, C].

```
Vanilla:
 SELECT timestamp FROM MessageLog
 WHERE msgType = "send" AND outcome = "failure"
 AND (receiver = "CompanyB" OR receiver = "CompanyA"
 OR receiver = "CompanyC")
 AND (level = [0,A] OR level = [0,B] OR level = [0,C]);

CW-DSMS [0,T]:
```

```
SELECT timestamp FROM MessageLog
WHERE msgType = "send" AND outcome = "failure"
AND (receiver = "CompanyB" OR receiver = "CompanyA"
OR receiver = "CompanyC");
```

The query language of CW-DSMS uses simplified form because of the replicated architecture. As shown in Table 9.7, the performance overhead is between 0.003% and 0.023%.

5. **Experiment 5: Cloud auditing in level [1, B]**:

```
SELECT MIN(timestamp), MAX(timestamp)
FROM MessageLog [ROWS 100]
WHERE outcome = "success" AND serviceId = "5";
```

We studied the overhead caused by the least upper bound computations in CW-DSMS. The output tuple level always reflects the highest possible level (COI class) of all the input tuples involved in the computation. Inputs were either at [1,⊥] or [⊥,B]. To maximize the difference, we used 100% selectivity. The performance difference due to LUB computation is between 0.011% and 0.030%.

6. **Experiment 6: Cloud auditing query 6 in level [1, B]**

```
SELECT R.timestamp - S.timestamp AS delay
FROM MessageLog R[Rows 100], MessageLog S[Rows 100]
WHERE S.msgType = "send" AND S.outcome = "success"
AND R.msgType = "receive" AND R.outcome = "success"
AND R.receiver = "Company1" AND R.sender = "CompanyB"
AND S.receiver = "CompanyB" AND S.sender = "Company1"
AND S.serviceId = R.serviceId;
```

In the join query, input stream R and S refer to the same input stream source MessageLog. We set up 50% selectivity for R and S respectively. To activate LUB computation on join, input tuples were kept at either [1, ⊥] or [⊥, B] and streamed in a random fashion.

Since join is an expensive operation, the input rate of 50,000 tuples used in the previous experiments caused an overload situation in both the systems. Thus, we reduced the data input rate to 2,500 tuples per second. Accordingly, the data sizes of the three input tuple sets were reduced to 100K, 250K, and 500K tuples, respectively. The performance overhead is between 0.081% and 0.218%, which is higher than the other experiments. On the other hand, the overhead is still considered negligible as it is within 0.218%.

# Chapter 10

# Conclusions and Future Work

## 10.1 Conclusions

Traditional DBMS is not sufficient to support real-time stream processing application in following reasons. (1) one-time SQL does not support continuous stream queries; (2) System cannot handle queries between stored table and live inputs such as streaming data; (3) Processing mechanism buffering first then execution causes unexpected storage cost and high latency.

Motivated by those real-time application needs, stream processing DSMSs have been developed to address continuous queries with unpredictable, massive input data. However, they do not provide security protections on many situation monitoring applications involve data that are classified at various security levels, such as battlefield monitoring, emergency threat, and resource management. Existing DSMSs must be redesigned to ensure that illegal information flow do not occur in such applications. Besides, the data stream management system should be able to deal with QoS requirements of multiple queries under the pressure of heavy input load.

Our goal is to develop a multilevel secure DSMS, which is able to provide security guarantee against illegal information follow, support flexible continuous queries with level classification, and execute multiple queries effectively by sharing and reuse mechanisms. Towards this end, this work includes the following contributions:

1. Provide systematic analysis on query language, process details, architecture design of a typical DSMS STREAM system, and discussions of limitations on its security preservation and

performance.

2. Formalize multilevel security model for data stream management system as well as the continuous query language. Continuous queries are able to support level-specific request.

3. Investigate possible system architectures with multilevel secure access control.

4. Develop replicated MLS-DSMS, which provides secure query scheduling and execution, as well as sharing mechanisms between queries in the same security level.

5. Develop trusted MLS-DSMS, which provides query rewriting and optimization before query generation, as well as sharing mechanisms between queries across different security levels.

6. Explore distributed system network, and propose group construction and load balancing algorithms.

7. Extend secure DSMS to support stream audit application, cooperated with Chinese Wall access control policy.

8. Implement the replicated, trusted, naive distributed MLS-DSMS and Chinese Wall DSMS prototypes. We've run experiments to study the overhead introduced by security properties, the performance gain from sharing mechanisms, and performance differences between trusted and replicated architectures.

## 10.2 Future Work

### 10.2.1 Security Label

In this work we take security level as a special attribute for every input tuple. Providing MLS control on different data granularity is one direction of our future work. If assigning security label

to some attributes instead of the whole tuple, there will be more than one security clearance in one data record. So users in different levels might be limited to access partial records and execute certain queries. For example, an accountant intern might be able to get the salary average of the company employees but cannot access individual record. In this case, we need to take care of the excessive privilege abuse threat [42]. Query access control list, which defines what queries are allowed against the table by specific user, is desirable.

### 10.2.2 More Sharing Consideration

Currently query sharing is the main approach to reduce the process time and resource usage. There are still some extended considerations.

- Suppose there are more than one sharable queries, how to select the best to reduce the number of operators in new queries plan.
- A new query is able to use sharing components from multiple queries.
- if the sharing cascade link is too long, what is the performance gain. For example, $Q_1$ is shared by $Q_2$, $Q_2$ is shared by $Q_3$, and so on, will $Q_n$ be sufficient compared with generating its plan without sharing?

Another direction is to find more sharing possibilities. An interesting topic is sharing in join operators using different input windows. Even though the computation operator can be identical, the difficulties are the synchronization of the computation content and the outputs. There are several reasons. First, the input order of arriving tuples are different between joins with window in different sizes, which might cause incorrect computation. Second, the expiration (negative sign) tuples are generated in different ways. Most stream computations will be triggered once receiving

expired inputs. As a result, the outputs might be changed if sharing other join results. Besides, extra order-correction mechanism might introduce storage overhead and high latency.

### 10.2.3 Prototype Development

Currently we have implemented centralized replicated and trusted MLS-DSMS prototypes with the functions like multilevel secure scheduling and execution, query rewriting and optimization, queries sharing in the same and across levels. We aim to add more mechanisms which are useful for MLS distributed system extension:

- We would like to introduce the simplify ordering and load shedding mechanisms developed in Aurora system [1] under distributed network. By such way part of the imperfection input situations like out-of-order/delay can be handled.

- We plan to introduce sophisticated encryption mechanisms for authentications between servers.

- We will investigate the possibilities of "execution rental service" between servers in different levels. Suppose all the servers belongs to group in classified level are under heavy-load, under what restrictions they can borrow servers from higher level like secret slaves to run their sanitized queries.

- The master-slave architecture is easy the manage but fragile if master is down. We can take the ideas of setting up back-up servers with k-safety guarantee, or develop a master delegation mechanism in the network.

- Our load distribution algorithm is based on CPU usage only in current stage. In high volume of input cases the limited storage is the bottleneck for machines. The distribution algorithm

can be extent with sophisticated to detect different cases of heavy load and provide smarter distribution.

- The current DSMS and the prototypes do not consider the user-specific constraints, which is popular in some real-time applications. For example, an investor requests urgent stock alert service like “Notify me in 10 seconds if the stock A’s price is lower than the history average in the market”. If the DSMS is designed to provide such service, it should put priority to this query any time to satisfy the response time constraint. User constraint is a useful extension in next MLS DSMS version.

### 10.2.4 Chinese Wall DSMS

A lot of work remains to be done. We have assumed that certain components are trusted. We have made similar assumptions about the underlying infrastructure. However, we have not explicitly stated our trust assumptions. We need to formally state and analyze these assumptions in view of real-world constraints in order to evaluate the security of our DSMS.

We plan to propose alternative architectures and do a comparative study to find out which approach is the most suitable for processing cloud streaming queries. We also plan to implement our query sharing ideas. Thus, when a new query is submitted, we need to check how plans for existing queries can be reused to improve the performance. Note that, such verification must be carried out dynamically. Towards this end, we plan to see how existing constraint solvers can be used to check for query equivalences. We also plan to evaluate the performance impact of dynamic plan generation and equivalence evaluation. We also plan to investigate more on how scheduling and load shedding can be done with information flow constraints.

# References


[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The Very Large Data Base Journal*, 12(2):120–139, August 2003.

[2] Marshall D. Abrams, Sushil G. Jajodia, and Harold. J. Podell, editors. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1995.

[3] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Transactions of Data Knowledge and Engineering*, 59(1):139–165, 2006.

[4] Raman Adaikkalavan and Thomas Perez. Secure Shared Continuous Query Processing. In *Proceedings of the ACM Symposium on Applied Computing (Data Streams Track)*, Taiwan, Mar 2011.

[5] Raman Adaikkalavan, Indrakshi Ray, and Xing Xie. Multilevel secure data stream processing. In *Proceedings of the 25th annual IFIP WG 11.3 conference on Data and applications security and privacy (DBSec'11)*, pages 122–137. Springer-Verlag, 2011.

[6] Raman Adaikkalavan, Xing Xie, and Indrakshi Ray. Multilevel secure data stream processing: Architecture and implementation. *Journal of Computer Security*, 20(5):547–581, 2012.

[7] Software AG. Webmethods business events. `http://www.softwareag.com/corporate/products/wm/events/capabilities/default.asp`.

[8] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, , and Jennifer Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[9] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic Foundations and Query Execution. *Very Large Data Base Journal*, 15(2):121–142, 2006.

[10] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 261–272, May 2000.

[11] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The Very Large Data Base Journal*, 13(4):333–353, 2004.

[12] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, June 2002.

[13] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 253–264, June 2003.

[14] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of International Conference on Data Engineering*, pages 350–361, March 2004.

[15] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *Very Large Data Base Journal: Special Issue on Data Stream Processing*, 13(4):370–383, 2004.

[16] David E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, The Mitre Corp., Burlington Road, Bedford, MA 01730, USA, March 1976.

[17] Elisa Bertino and Ravi Sandhu. Database security – concepts, approaches, and challenges. *IEEE TRANS. DEPENDABLE SECUR. COMPUT*, 2(1):2–19, 2005.

[18] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Ankush Gupta, Laura M. Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, Ying Yan, Beomjin Yun, and Jin Zhang. A demonstration of the maxstream federated stream processing system. In *Proceedings of International Conference of Data Engineering*, pages 1093–1096, 2010.

[19] David F. C. Brewer and Michael J. Nash. The Chinese Wall Security Policy. pages 206–214, May 1989.

[20] Jianneng Cao, Barbara Carminati, Elena Ferrari, and Kian-Lee Tan. Acstream: Enforcing access control over data streams. *Data Engineering, International Conference on*, 0:1495–1498, 2009.

[21] Barbara Carminati, Elena Ferrari, and Kian Lee Tan. Enforcing access control over data streams. In *Proc. of the ACM SACMAT*, pages 21–30, 2007.

[22] Don Carney, Ugur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. *Proceedings of Very Large Data Base*, pages 838–849, August 2003.

[23] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 215–226, August 2002.

[24] Silvana Castano, Maria Grazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security (ACM Press Book)*. Addison-Wesley, 1994.

[25] Sharma Chakravarthy and Raman Adaikkalavan. Event and Streams: Harnessing and Unleashing Their Synergy. In *International Conference on Distributed Event-based Systems*, pages 1–12, July 2008.

[26] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Advances in Database Systems , Vol. 36. Springer, 2009.

[27] Sharma Chakravarthy and Vamshi Pajjuri. Scheduling Strategies and Their Evaluation in a Data Stream Management System. In *BNCOD*, pages 220–231, 2006.

[28] Fa-Chung Fred Chen and Margaret H. Dunham. Common Subexpression Processing in Multiple-Query Processing. *IEEE Transactions on Knowledge and Date Engineering*, 10(3):493–499, 1998.

[29] Yi cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load shedding in stream databases: a control-based approach. In *Processings International Conference on Very Large Data Base*, pages 787–798, 2006.

[30] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *Conference on Innovative Data Systems Research*, 2003.

[31] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR*

*2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[32] Committee on Multilevel Data Management Security, Air Force Studies Board, Commission on Engineering and Technical Systems, National Research Council, National Academy Press, Washington D.C. *Multilevel data management security*, March 1983.

[33] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 40–51, June 2003.

[34] Charlies Donnelly and Richard M. Stallman. *Bison Manual for Version 1.875: Using the YACC-compatible Parser Generator*. Free Software Foundation, 2003.

[35] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. *SIGOPS Operating System Review*, 42:301–313, April 2008.

[36] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39:17–30, October 2005.

[37] S. J. Finkelstein. Common Subexpression Analysis in Database Applications. In *Proceedings, International Conference on Management of Data (SIGMOD)*, pages 235–245, June 1982.

[38] Binto George and Jayant R. Haritsa. Secure Concurrency Control in Firm Real-Time Databases. *Distributed and Parallel Databases*, 5:275–320, 1997.

[39] Jonathan Goldstein and Per A. Larson. Optimizing queries using materialized views. In *Proceedings, International Conference on Management of Data (SIGMOD)*, pages 331–342, June 2001.

[40] IBM. Infosphere system 3.0, 2012. `http://www.ibm.com/software/data/infosphere/streams/`.

[41] IBM Stream Processing. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html`.

[42] Impreva. Top ten database threats: How to mitigate the most significant database vulnerabilities, February 2011. `http://www.globalsecuritymag.com/IMG/pdf/WP_TopTen_Database_Threats.pdf`.

[43] Trent Jaeger, Reiner Sailer, and Yogesh Sreenivasan. Managing the risk of covert information flows in virtual machine systems. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 81–90, New York, NY, USA, 2007. ACM.

[44] Qingchun Jiang and Sharma Chakravarthy. Scheduling strategies in a data stream management system. Technical report, UT Arlington, 2003.

[45] Qingchun Jiang and Sharma Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *21st Annual British National Conference on Databases*, pages 16–30, July 2004.

[46] Qingchun Jiang and Sharma Chakravarthy. Anatomy of a Data Stream Management System. In *ADBIS Research Communications*, 2006.

[47] Chun Jin and Jaime Carbonell. Predicate indexing for incremental multi-query optimization. In *Proceedings of the 17th international conference on Foundations of intelligent systems*, ISMIS'08, pages 339–350, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] Balakumar Kendai and Sharma Chakravarthy. Load Shedding in MavStream: Analysis, Implementation, and Evaluation. In *Proceedings, International British National Conference on Databases (BNCOD)*, pages 100–112, 2008.

[49] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *SIGOPS Operating System Review*, 41:321–334, October 2007.

[50] Wolfgang Lindner and Jörg Meier. Securing the borealis data stream engine. In *Proceedings of the 10th International Database Engineering and Applications Symposium (IDEAS)*, pages 137–147, 2006.

[51] Matteo Migliavacca, Ioannis Papagiannis, David M. Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. Defcon: high-performance event processing with information security. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[52] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation. In *Conference on Innovative Data Systems Research*, pages 245–256, 2003.

[53] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[54] Rimma V. Nehme, Hyo-Sang Lim, Elisa Bertino, and Elke A. Rundensteiner. StreamShield: A stream-centric approach towards security and privacy in data stream environments. In *proceedings of ACM SIGMOD*, pages 1027–1030, 2009.

[55] Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. A security punctuation framework for enforcing access control on streaming data. In *Proceedings of International Conference of Data Engineering*, pages 406–415, 2008.

[56] Gultekin Ozsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: a survey. *Knowledge and Data Engineering, IEEE Transactions on*, 7(4):513 –532, August 1995.

[57] Satya Kiran Popuri. Understanding c parsers generated by gnu bison, 2006. `www.cs.uic.edu/~spopuri/cparser.html`.

[58] Ravi Sandhu. Lattice-Based Enforcement of Chinese Walls. 11(8):753–763, 1992.

[59] Timos K. Sellis. Multi-Query Optimization. *ACM Transactions on Database Systems Journal*, 13(1):23–52, 1988.

[60] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Advances in Database Systems , Vol. 36. Springer. ISBN 978-0-387-71002-0, 2009.

[61] Sang H. Son and Rasikan David. Design and analysis of a secure two-phase locking protocol. In *Computer Software and Applications Conference (COMPSAC 94), Eighteenth Annual International*, pages 374 –379, November 1994.

[62] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the 23th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274, 2004.

[63] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, December 2005.

[64] StreamBase. `http://www.streambase.com`.

[65] StreamBase. Streambase CEP platform. `http://www.streambase.com/products/streambasecep`.

[66] Sybase. Sybase Aleri Streaming Platform Home Page, 2010. `http://m.sybase.com/products/financialservicessolutions/complex-event-processing`.

[67] Nesime Tatbul, Ugur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *proceedings of Very Large Data Base*, pages 309–320, September 2003.

[68] Nesime Tatbul and Stanley B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of Very Large Data Base*, pages 799–810, 2006.

[69] STREAM Team. Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab, 2003.

[70] STREAM Team. *STREAM Prototype Source Code*. Stanford University, 2005. `http://infolab.stanford.edu/stream/code/`.

[71] Tien-Hao Tsai, Yen-Chung Chen, Hsiu Huang, Pei-Ming Huang, and Kuo-Sen Chou. A practical chinese wall security model in cloud computing. pages 1–4, September 2011.

[72] Stratis D. Viglas and Jeffery F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 37–48, June 2002.

[73] Ruoyu Wu, Gail-Joon Ahn, Hongxin Hu, and Mukesh Singhal. Information flow control in cloud computing. pages 1–7, October 2010.

[74] Rui Xie and Rose Gamble. A Tiered Strategy for Auditing in the Cloud. In *IEEE International Conference on Cloud Computing*, June 2012.

[75] Xing Xie, Indrakshi Ray, Raman Adaikkalavan, and Rose Gamble. Information flow control for stream processing in clouds. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, Amsterdam, The Netherlands, June 2013. To appear.

[76] Xing Xie, Indrakshi Ray, Waruna Ranasinghe, Philips Gilbert, Pramod Shashidhara, and Anoop Yadav. Distributed multilevel secure data stream processing. In *Proceedings of the 12th International Workshop on Assurance in Distributed Systems and Networks*, Philadelphia,USA, July 2013. To appear.

[77] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.